Содержание

Введение

О чём это

Программировать — сложно.

Хороший код адекватно отражает систему, которую описывает, он устойчив к изменениям в этой системе. Плохой код запутанный, хрупкий и непонятный — он замедляет разработку.

Код становится плохим, когда он перестаёт соответствовать реальности — бизнес-требованиям, правилам поведения частей системы, их отношениям друг с другом.

Бизнес-правила — это территория. Код — карта этой территории. Чем точнее карта, тем проще справляться с изменениями в требованиях и даже предвидеть их.

В этой книге мы хотим рассказать и показать на примерах, как принципы объектно-ориентированного программирования могут помочь спроектировать устойчивую систему.

Почему ООП?

Объектно-ориентированное программирование, ООП, вызывает споры.

Война парадигм может создать впечатление, что подход ООП устарел и плох. Но ООП — это всего лишь инструмент. У каждого инструмента есть удобства, недостатки и область применения. Инструмент должен оставаться способом решения задачи и не должен становиться самоцелью.

ООП, как инструмент, может помочь спроектировать систему на языке, более близком к языку бизнес-правил. Это снижает вероятность ошибки при переводе с «языка бизнеса» на «язык разработки» и наоборот.

О каких принципах пойдёт речь?

Мы рассмотрим 5 принципов SOLID, а именно:

- принцип единственной ответственности (single responsibility principle);
- открытости и закрытости (open/closed principle);
- подстановки Барбары Лисков (Liskov substitution principle);
- разделения интерфейса (interface segregation principle);
- инверсии зависимостей (dependency inversion principle).

Каждый из них — это лишь рекомендация, все они имеют область и границы применения. Но чтобы увидеть эти границы, необходимо понять, в чём польза и издержки каждого.

Многие принципы вам покажутся чрезмерно абстрактными, неконкретными или вовсе надуманными. Отнеситесь к таким принципам, как к дзену Python — держите в голове, но проверяйте, насколько они полезны в конкретной ситуации.

Мы в этой книге предлагаем ещё одну из бесконечных интерпретаций этих принципов, попутно расписывая пользу и ограничения каждого. Зная пользу и ограничения, можно оценить, насколько конкретный принцип помогает решить задачу, стоящую перед вами.

Какой план?

Каждый раздел будет описывать один из принципов и показывать, как им пользоваться в повседневной работе.

Мы будем рассматривать примеры на TypeScript, так как в нём есть понятия, которые нам пригодятся по ходу повествования. Если вы чувствуете себя неуверенно с TypeScript, попробуйте прочесть книгу TypeScript Deep Dive — она содержит все концепции, которые мы будем использовать.

В конце разделов вы найдёте проверочные вопросы. Каждый правильно отвеченный вопрос увеличивает количество очков на вашем счету. Максимально-возможный счёт — 100 очков. Отвечайте аккуратно — вопросы с подвохом. Обратите внимание, вариантов ответа может быть больше одного.

Материалы к разделу

- Объектно-ориентированное программирование
- Bob Martin SOLID Principles of Object Oriented and Agile Design, Youtube
- Карта ≠ территория
- TypeScript Deep Dive

SRP

Введение

Чтобы вникнуть в этот и другие принципы, нам понадобится ввести несколько понятий, которые будем использовать.

Модулем мы будем называть какую-то часть кода, обособленную от других. Это может быть класс, функция, объект, файл — в общем, что-то, у чего есть границы, отделяющие этот код от другого.

Бизнес-правилами будем называть правила взаимодействия сущностей друг с другом и внешней средой. Под внешней средой будем понимать всё, что влияет на программу извне — пользовательский ввод, события, вызов API и т.д.

Причиной изменения будем называть обновление бизнес-правил, которое вынуждает менять код какого-либо модуля.

Принцип единственной ответственности

Принцип единственной ответственности (Single Responsibility Principle, SRP) означает, что у модуля должна быть только одна причина для изменения. Весь код, который меняется по этой причине, должен быть собран в этом модуле.

Проще говоря принцип предлагает нам проводить границы между модулями так, чтобы изменение в бизнес-правилах затрагивало как можно меньше модулей, в идеале — один.

Основной инструмент принципа — объединять те части, которые меняются по одной причине, и разделять те, которые меняются по разным.

Принцип позволяет уменьшить количество кода, который нужно менять при изменении бизнес-правил. Он помогает ограничить влияние этих изменений и контролировать сложность программы.

Коротко

Принцип единственной ответственности:

- помогает разбивать и декомпозировать задачи по одной на модуль;
- уменьшает количество модулей, которые надо изменить при изменении требований;
- ограничивает влияние изменений, помогая контролировать сложность системы.

Материалы к разделу

- Think you understand SRP?
- SRP: Не такой простой, как кажется
- Принцип единственной ответственности
- SOLID: Часть 1, SRP

В идеальном мире

В идеальном мире каждый класс в коде решает одну и только одну задачу, а все задачи структурированы и разделены. Модули в этом случае дополняют друг друга, а их совокупность детально описывает систему.

Допустим, у нас есть задача создать отчёт об активности пользователей и вывести его в нескольких вариантах: как строку HTML или TXT.

Отчёт

Мы создадим класс `ReportExporter`, который будет заниматься только экспортом данных. Определять необходимый формат будет класс `FormatSelector`. А форматированием данных будут заниматься классы: `HtmlFormatter` и `TxtFormatter`.

```
// Тип данных для отчёта:

type ReportData = {
  content: string,
  date: Date,
  size: number,
}

// Возможные форматы:

enum ReportTypes {
  Html,
  Txt,
}

// Класс, который занимается экспортом данных:

class ReportExporter {
  name: string
  data: ReportData

  constructor(name: string, data: ReportData) {
    this.name = name
    this.data = data
  }

  export(reportType: ReportTypes): string {
    const formatter: Formatter = FormatSelector.selectFor(reportType)
    return formatter.format(this.data)
  }
}
```

Форматы экспорта

В соответствии с SRP форматирование данных — это *отдельная задача*. Поэтому для преобразования данных отчёта в необходимый формат мы создадим отдельные классы.

```
interface Formatter {
  format(data: ReportData): string
}

// Класс для форматирования в HTML:
```

```
class HtmlFormatter implements Formatter {
  format(data: ReportData): string {

    // ...Форматируем данные в HTML и возвращаем:
    return 'html string'
  }
}

// Класс для форматирования в TXT:
class TxtFormatter implements Formatter {
  format(data: ReportData): string {

    // ...Форматируем данные в TXT и возвращаем:
    return 'txt string'
  }
}
```

Выбор формата

Принцип единственной ответственности подсказывает, что выбор формата не входит ни в задачу форматирования данных, ни в задачу их подготовки. Поэтому существующие классы нам не подойдут.

Для решения этой задачи воспользуемся шаблоном проектирования «Стратегия», который поможет выбрать подходящий формат. (Более подробно «Стратегию» мы разберём в разделе о принципе открытости и закрытости.)

Создадим новый класс `FormatSelector`, который будет выбирать тип форматирования, в зависимости от настроек.

```
class FormatSelector {
  private static formatters = {
    [ReportTypes.Html]: HtmlFormatter,
    [ReportTypes.Txt]: TxtFormatter,
  }

  static selectFor(reportType: ReportTypes) {
    const FormatterFactory = FormatSelector.formatters[reportType]
    return new FormatterFactory();
  }
}

const dynamicFormatter = FormatSelector.selectFor(ReportTypes.Html)
dynamicFormatter.format(/*...*/)
```

Таким образом SRP помогает разделить ответственность за различные задачи между сущностями и сделать это так, чтобы каждая сущность занималась одной задачей.

В реальной жизни

В реальной жизни SRP также позволяет решать задачи эффективнее. Ниже мы привели несколько примеров из проектов, над которыми работали.

Валидация форм

В вебе одна из распространённых задач — валидация форм.

Часто разработчики валидируют данные прямо в контроллерах форм. Это приводит к тому, что код контроллеров становится чрезмерно объёмным и нечитаемым. Также такой подход грозит дублированием кода, когда одинаковая валидация используется в разных формах.

По принципу единственной ответственности следует разделять код, который меняется по разным причинам. В случае с валидацией формы есть две зоны ответственности.

Первая — представление данных: вывод полей, значений, прогресса заполненности формы, ошибок и прочее. Вторая — преобразование и обработка данных. Валидация попадает во вторую.

В идеале валидация не должна ничего знать о том, в каком виде данные выводятся. Она должна работать только со значениями и принимать наборы данных в обговорённом формате, не привязываясь к форме вовсе.

Применение SRP для валидации позволяет:

- уменьшить код обработчиков форм, выделив валидацию в отдельный модуль;
- держать валидацию значений в одном месте, собрав всё, что к ней относится в одном модуле;
- валидировать не только формы, а любые наборы данных, которые соответствуют обговорённому формату валидатора.

Обработка сокет-событий

Для работы с данными в реальном времени в вебе часто используется socket.io.

Если в системе есть компоненты, которые каким-то образом работают с сокет-событиями, есть соблазн обрабатывать эти события прямо внутри компонента. На первый взгляд это даже не противоречит принципу: ведь события непосредственно связаны с компонентом и задачей, которую он решает.

Но события в компоненте — это не обязательно сокет-события. У компонента по-хорошему должен быть интерфейс, который бы описывал события компонента и его поведение. (Подробнее об интерфейсах мы поговорим в разделе о принципе разделения интерфейса.)

По SRP настройку работы именно с сокетами следует вынести в отдельный модуль, причиной изменения которого будет только настройка и зависимость сокетов.

Настройка бюджета во второй версии Тяжеловато

Тяжеловато — это приложение, которое помогает экономить. Пользователи указывают, сколько денег у них есть и на какой срок они хотят их растянуть. Приложение рассчитывает бюджет и сумму на день.

В первой версии бюджет и сумма на день были частями одной сущности — бюджета. Это сильно усложняло расчёты и увеличивало объём обработчиков пользовательских событий.

По принципу единственной ответственности работа с суммой на день и с общим бюджетом на весь период — разные задачи. Причина изменения суммы на день — ввод траты; причина изменения бюджета — изменение настроек бюджета.

В обновлённой версии бюджет и сумма на день стали отдельными сущностями, которые общаются друг с другом через сообщения и команды.

Это позволило:

- уменьшить количество эдж-кейсов при расчётах сумм;
- сделать общение между сущностями более прозрачным;
- разделить ответственность за обработку трат и настроек.

Шаблоны проектирования и приёмы рефакторинга

Соблюдать принцип единственной ответственности позволяют несколько шаблонов проектирования и приёмов рефакторинга.

Выделение класса

Выделение класса — приём рефакторинга, при котором из большого класса с множеством слабо-связанных по смыслу полей и методов, выделяется один или несколько классов.

Смысл приёма в том, чтобы явно выделить назначение класса. Идеальный результат — получить класс, который можно описать одной фразой или даже одним словом.

В примере ниже до рефакторинга мы имеем класс `Person`, который содержит логику преобразования телефонного номера. После — эта функциональность вынесена в класс `PhoneNumber`.

```
// До рефакторинга:

class Person {
  name: string
  phone: string
  officeCode: string

  constructor(name: string, phone: string, officeCode: string) {
    this.name = name
    this.phone = phone
    this.officeCode = officeCode
  }

  phoneNumberOf(): string {
    return `${this.phone} доб. ${this.officeCode}`
  }
}

// После:

interface IPhoneNumber {
  phone: string
  officeCode: string
  valueOf(): string
}
```

```
class PhoneNumber implements IPhoneNumber {
  phone: string
  officeCode: string

  constructor(phone: string, officeCode: string) {
    this.phone = phone
    this.officeCode = officeCode
  }

  valueOf(): string {
    return `${this.phone} доб. ${this.officeCode}`
  }
}

class Person {
  name: string
  phoneNumber: IPhoneNumber

  constructor(name: string, phoneNumber: IPhoneNumber) {
    this.name = name
    this.phoneNumber = phoneNumber
  }

  phoneNumberOf(): string {
    return this.phoneNumber.valueOf()
  }
}
```

Класс `Person` теперь работает только с данными пользователя, а задача преобразования номера делегируется экземпляру класса `PhoneNumber` через зависимость в конструкторе.

Фасад

Фасад — шаблон проектирования, при котором сложная логика скрывается за вызовом более простого API.

Фасад обеспечивает простое общение со сложной частью системы, беря ответственность за настройку и вызов специфических методов конкретных объектов на себя.

Один из минусов фасада в том, что он может превратиться в божественный объект.

В примере ниже мы выносим инициализацию и настройки классов `Square` и `Circle` в фасад `ShapeFacade`. После этого мы можем вызывать метод `.areaOf` фасада и получать площадь любой фигуры, которая подготовлена внутри фасада.

```
class Square extends Figure {
  length: number

  constructor(length: number) {
    this.length = length
  }
```

```
  areaOf(): number {
    return this.length ** 2
  }
}

class Circle extends Figure {
  radius: number

  constructor(radius: number) {
    this.radius = radius
  }

  areaOf(): number {
    return Math.PI * (this.radius ** 2)
  }
}

// Применение «Фасада»:

class ShapeFacade {
  square: Square
  circle: Circle

  constructor() {
    this.square = new Square(42)
    this.circle = new Circle(42)
  }

  areaOf(figure: string): number {
    switch (figure) {
      case 'square': return this.square.areaOf()
      case 'circle': return this.circle.areaOf()
      default: return 0
    }
  }
}
```

Прокси

Прокси — шаблон проектирования, при котором общение с каким-то объектом контролирует другой объект-заместитель (прокси). Он позволяет расширять функциональность существующих классов, не меняя их.

В примере мы используем прокси `LoggedRequest`, чтобы не примешивать логирование в класс, который реализует запросы к серверу `RequestClient`.

```
class RequestClient {
  async request(url: string): Promise<any> {
    try {
      const response = await fetch(url)
      const data = await response.json()
      return data
```

```
    }
    catch (e) {
      return null
    }
  }
}

class LoggedRequest {
  client: RequestClient

  constructor(client: RequestClient) {
    this.client = client
  }

  async request(url: string): Promise<any> {
    console.log(`Performed request to ${url}`)
    return await this.client.request(url)
  }
}
```

Материалы к разделу

- Выделение класса
- Пример использования выделения класса
- Фасад
- Пример использования фасада
- Реализация фасада на TypeScript
- Божественный объект
- Прокси
- Пример использования прокси
- Реализация прокси на TypeScript

Антипаттерны и запахи

Существуют и антипаттерны, которые или сами нарушают SRP, или усложняют следование принципу.

Божественный объект

Божественный объект — это сущность, которая отвечает за слишком много.

Проблема таких объектов в том, что внутри них скапливается неоправданно большое количество данных. Со временем может случиться, что никакое действие нельзя будет сделать без участия божественного объекта.

Это затрудняет рефакторинг, тестирование и внесение изолированных изменений в код.

Синглтон

Синглтон — это паттерн, при котором в приложении существует только один экземпляр какого-то класса. Существующий синглтон гарантирует, что все новые созданные объекты будут ссылаться на него.

С точки зрения SRP это смешение ответственностей. Потому что синглтон не только выполняет свою основную функцию, но ещё и проверяет, не существует ли уже созданных экземпляров.

Проблем у этого паттерна несколько:

- он глобален — когда нарушается инкапсуляция состояния, повышается вероятность непредсказуемых нежелательных изменений;
- излишне имплицитен — трудно заранее понять, как себя поведёт объект в какой-то ситуации;
- трудно тестируется — глобальный объект хранит большое количество данных и может находиться в большом количестве различных состояний, из-за чего модульные тесты могут показывать непредсказуемые результаты.

Смешение архитектурных слоёв

Паттерн Model-View-Controller, MVC подразумевает три сущности: модель, представление и контроллер. Модель отвечает за хранение данных; представление — за их отображение; контроллер — за преобразование и обработку.

Смешение слоёв — это неправильное распределение или размазывание ответственностей между слоями. Оно может приводить:

- к разрастанию контроллера, что делает его код трудным для понимания;
- появлению логики внутри модели, что затрудняет переиспользование модели.

Материалы к разделу

- 3 способа нарушить SRP
- Божественный объект
- God object. Анализ сложных проектов
- Синглтон
- Паттерн MVC
- Why to avoid fat controllers

Ограничения и подводные камни

У каждого принципа есть ограничения и область применения. Для SRP характерны следующие ограничения и подводные камни.

Слепое следование опасно

Слепое следование принципу (например, преждевременная оптимизация) может создать лишние абстракции и сделать программу чрезмерно сложной. Использовать SRP стоит тогда, когда вносить изменения в класс, нарушающий его, становится дорого.

Трудность начального проектирования

Выделить зоны ответственности сущностей при первичном проектировании иногда бывает трудно. Взаимодействие сущностей может быть неочевидным, из-за чего проектировщикам может быть трудно определиться с методами классов. В этом случае могут помочь прототипирование и диаграммное моделирование:

- Unified Modeling Language, UML;
- Data flow diagram, DFD;
- Entity relationship diagram, ERD;

- Методология моделирования IDEF.

Разработка через тестирование — не панацея, а инструмент

Считается, что разработка через тестирование (Test driven development, TDD) может помочь выделить в классах необходимые методы, но это тоже помогает не всегда. TDD определённо показывает, каким будет видеть наше API пользователь, но он не всегда помогает выделить зоны ответственности модулей.

Материалы к разделу

- Унифицированный язык моделирования, UML
- Диаграммы потоков данных, DFD
- Entity-relationship модель
- Методология моделирования, IDEF
- Разработка через тестирование, TDD
- TDD на примере

OCP

Введение

Проектируя систему, мы занимаемся моделированием, а значит решаем инженерную задачу. Мы строим гипотезу о том, каковы отношения между сущностями в этой системе.

Однако бизнес-требования не вечны, они могут (и будут) меняться. Хорошо спроектированная система способна пережить эти изменения, отразить их в себе и продолжить функционировать.

Основная причина, по которой вносить изменения бывает трудно или дорого — когда небольшое изменение в одной части системы вызывает лавину изменений в других частях. Грубо и утрировано: если в программе для изменения цвета кнопки надо поправить 15 модулей, такая система спроектирована плохо.

Принцип открытости-закрытости

Принцип открытости-закрытости (Open-Closed Principle, OCP) помогает исключить такую проблему. Согласно ему модули должны быть открыты для расширения, но закрыты для изменения.

Простыми словами — модули надо проектировать так, чтобы их требовалось менять как можно реже, а расширять функциональность можно было с помощью создания новых сущностей и композиции их со старыми.

Основная цель принципа — помочь разработать проект, устойчивый к изменениям, срок жизни которых превышает срок существования первой версии проекта.

Модули, которые удовлетворяют OCP:

- *открыты для расширения* — их функциональность может быть дополнена с помощью других модулей, если изменятся требования;
- *закрыты для изменения* — расширение функциональности модуля не должно приводить к изменениям в модулях, которые его используют.

Конечно, всегда есть изменения, которые невозможно внести, не изменив код какого-то модуля — никакая система не может быть закрыта на 100%. Поэтому при проектировании важен стратегический подход. Необходимо определить, от каких именно изменений и какие именно модули вы хотите закрыть. Это решение следует принимать опираясь на опыт, а также знания предметной области и пользователей системы.

Нарушение принципа открытости-закрытости приводит к ситуациям, когда изменение в одном модуле вынуждает менять другие, связанные с ним. Это в свою очередь нарушает принцип единой ответственности, SRP, потому что весь код, который меняется по какой-то одной причине, должен быть собран в одном модуле. (Разные модули — разные причины для изменения.)

На примере

На схеме ниже объект `Client` непосредственно связан с объектом `Server`. Если нам вдруг понадобится, чтобы `Client` мог работать с разными объектами `Server`, нам придётся поменять его код.

Чтобы решить эту проблему, необходимо связывать объекты не напрямую, а через абстракции. Если все объекты `Server` реализуют интерфейс `Abstract Server`, то нам уже не придётся менять код объекта `Client` для замены одного объекта `Server` на другой.

В коде

Если попробовать выразить этот принцип в коде, то самым простым примером будет замена множественных проверок на принадлежность к типу на абстракцию.

Например, мы пишем сервис рассылки сообщений. Он принимает текст, который надо выслать, и сервис стороннего API для отправки СМС, пушей или электронных писем. Плохо спроектированный сервис мог бы выглядеть так:

```
class SmsSender {
  sendSms(message: MessageText) { /* ... */ }
}

class PushSender {
  sendPush(message: MessageText) { /* ... */ }
}

class EmailSender {
  sendEmail(message: MessageText) { /* ... */ }
}

class Notifier {
  constructor(private api: SmsSender | PushSender | EmailSender) {}

  notify(): void {
    const message = 'Some user notification';

    if (this.api instanceof SmsSender) {
      this.api.sendSms(message)
    } else if (this.api instanceof PushSender) {
      this.api.sendPush(message)
    } else if (this.api instanceof EmailSender) {
      this.api.sendEmail(message)
    }
  }
}
```

Проблема этого кода в том, что при *добавлении* нового типа стороннего API — голубиной почты — нам придётся *менять* уже существующий код.

```
// ...Предыдущие классы.

// Добавили новый тип стороннего API:

class DoveSender {
  sendDove(message: MessageText) { /* ... */ }
}

class Notifier {
  constructor(private api: SmsSender | PushSender | EmailSender | DoveSender) {}

  notify(): void {
```

```
    const message = 'Some user notification';

    if (this.api instanceof SmsSender) {
      this.api.sendSms(message)
    } else if (this.api instanceof PushSender) {
      this.api.sendPush(message)
    } else if (this.api instanceof EmailSender) {
      this.api.sendEmail(message)
    } else if (this.api instanceof DoveSender) { // Последние 3 строчки —
      this.api.sendDove(message)                 // ...это новый код,
    }                                            // ...который пришлось добавить.
  }
}
```

Вместо этого

OCP же предлагает не проверять конкретные типы, а использовать абстракцию, которая позволит не менять код класса `Notifier`. Для этого мы создадим интерфейс `Sender`, который будут реализовывать классы `SmsSender`, `PushSender` и `EmailSender`:

```
// Интерфейс будет абстракцией, которая описывает контракт,
// по которому должны работать классы, реализующий этот интерфейс:

interface Sender {
  sendMessage(message: MessageText): void
}

class SmsSender implements Sender {
  sendMessage(message: MessageText) {
    /* То, что раньше было внутри метода `sendSms`. */
  }
}

class PushSender implements Sender {
  sendMessage(message: MessageText) {
    /* То, что раньше было внутри метода `sendPush`. */
  }
}

class EmailSender implements Sender {
  sendMessage(message: MessageText) {
    /* То, что раньше было внутри метода `sendEmail`. */
  }
}
```

Тогда классу `Notifier` перестанет быть нужно проверять конкретный тип, и он сможет положиться на контракт, описанный в интерфейсе:

```
class Notifier {
  constructor(private api: Sender) {}
```

```
  notify(): void {
    const message = 'Some user notification';
    this.api.sendMessage(message);
  }
}
```

В результате

Теперь при *добавлении* голубиной почты, нам уже не потребуется *менять* код класса `Notifier`, зависящего от интерфейса `Sender`:

```
class DoveSender implements Sender {
  sendMessage(message: MessageText) {
    /* То, что раньше было внутри метода `sendDove`. */
  }
}

// Код класса `Notifier` останется тем же.
```

Таким образом, вводя адекватную абстракцию мы «расцепляем» модули. Мы делим зоны ответственности между разными частями приложения и уменьшаем количество кода, который нужно изменять при добавлении новой функциональности.

Коротко

Принцип открытости-закрытости:

- заставляет проектировать модули так, чтобы они делали только одну вещь и делали её хорошо;
- побуждает связывать сущности через абстракции (а не реализацию) там, где могут поменяться бизнес-требования;
- обращает внимание проектировщиков на места стыка и взаимодействие сущностей;
- позволяет сократить количество кода, который необходимо менять при изменении бизнес-требований;
- делает внесение изменений безопасным и относительно дешёвым.

Материалы к разделу

- Принцип открытости-закрытости
- Open-closed principle, Duke computer science
- Clarifying my use of the open/closed principle, Eric Elliott
- Научный метод
- Основы теории решения изобретательских задач

В идеальном мире

В идеальном мире при изменении бизнес-требований код модулей менять не приходится, а для реализации новых требований достаточно добавить новую сущность.

Нарушение OCP и конкретика

Ключ к пониманию OCP — *применение абстракций в местах стыка модулей*. Рассмотрим пример: требуется написать программу, которая будет считать площади фигур на экране.

Допустим, у нас есть класс прямоугольника `Rectangle`:

```
class Rectangle {
  width: number
  height: number

  constructor(width: number, height: number) {
    this.width = width
    this.height = height
  }
}
```

Следуя SRP подсчёт площади всех фигур мы вынесем в отдельный класс `AreaCalculator`. Вначале напишем его, не следуя принципу открытости-закрытости:

```
class AreaCalculator {
  shapes: Rectangle[]

  constructor(shapes: Rectangle[]) {
    this.shapes = shapes
  }

  totalAreaOf(): number {
    return this.shapes.reduce((tally: number, shape: Rectangle) => {
      return tally += (shape.width * shape.height)
    }, 0)
  }
}
```

Проблема в том, что если придётся добавить новую фигуру, например, круг, то для правильной работы, необходимо будет изменить и код класса `AreaCalculator`.

```
class Circle {
  radius: number

  constructor(radius: number) {
    this.radius = radius
  }
}

class AreaCalculator {
  // 1. Приходится менять тип:
  shapes: [Rectangle|Circle]

  constructor(shapes: [Rectangle|Circle]) {
    this.shapes = shapes
  }
```

```
  totalAreaOf(): number {
    return this.shapes.reduce((tally: number, shape: Rectangle | Circle) => {
      // 2. Приходится проверять, какой тип,
      //    чтобы применить правильный расчёт:
      if (shape instanceof Rectangle) {
        return tally += (shape.width * shape.height)
      }
      else if (shape instanceof Circle) {
        return tally += (shape.radius ** 2 * Math.PI)
      }
      else return tally
    }, 0)
  }
}
```

И подобные изменения придётся проводить для каждой новой фигуры.

Основной индикатор проблемы с принципом открытости-закрытости — появление проверки на `instanceof`. Если внутри кода модуля проверяется реализация, значит модуль жёстко привязан к другому, и изменения в требованиях заставят менять код этого модуля.

Применение OCP и абстракции

Чтобы исправить ситуацию, свяжем модули через абстракцию. Создадим интерфейс `AreaCalculatable`, который будет описывать поведение любой фигуры в системе, площадь которой можно посчитать.

```
interface AreaCalculatable {
  areaOf(): number
}
```

Это по сути ограничение на поведение сущностей внутри системы — гипотеза того, как они друг с другом взаимодействуют. В целом люди плохо умеют прогнозировать и предсказывать. И хотя опытный проектировщик, имея достаточно знаний о проектируемой системе, может сделать хорошее предположение, OCP всё же предлагает методику Just-in-time design. Она предполагает внесение изменений и добавление сущностей по мере необходимости, но не раньше.

Вернёмся к примеру. Сейчас классы фигур подчиняются новому ограничению и реализуют интерфейс `AreaCalculatable`:

```
// 1. Указываем, что класс реализует интерфейс,
//    это задаст ограничение на методы класса:

class Rectangle implements AreaCalculatable {
  width: number
  height: number

  constructor(width: number, height: number) {
    this.width = width
    this.height = height
  }
```

```
  // 2. Без этого метода класс считается не готовым,
  //    он же позволит абстрагироваться от реализации конкретной фигуры:

  areaOf(): number {
    return this.width * this.height
  }
}

// Те же изменения проводим для круга:

class Circle implements AreaCalculatable {
  radius: number

  constructor(radius: number) {
    this.radius = radius
  }

  areaOf(): number {
    return Math.PI * (this.radius ** 2)
  }
}
```

Теперь, когда у классов есть ограничения и правила, мы можем применить абстракцию, чтобы привязать их к `AreaCalculator`:

```
class AreaCalculator {
  // 1. Теперь тип абстрактный,
  //    мы можем указывать какие угодно фигуры
  //    при условии, что они реализуют `AreaCalculatable`:
  shapes: AreaCalculatable[]

  constructor(shapes: AreaCalculatable[]) {
    this.shapes = shapes
  }

  totalAreaOf(): number {
    return this.shapes.reduce((tally: number, shape: AreaCalculatable) => {
      // 2. Никаких проверок на классы, только вызов `areaOf`.
      //    Если даже мы добавим треугольник,
      //    нам не придётся менять код калькулятора:
      return tally += shape.areaOf()
    }, 0)
  }
}
```

Материалы к разделу

- Open-closed principle, Duke CS, PDF
- A simple example of the Open/Closed Principle
- Understanding SOLID: Open Closed Principle
- The OCP and what hides behind it

- Just-in-time design

В реальной жизни

Принцип открытости-закрытости побуждает исследовать отношения между сущностями до того, как вы начнёте писать код. Это помогает выявлять ошибки проектирования на ранних этапах.

Кроме этого OCP помогает отвязать модули друг от друга. Это минимизирует количество модулей, которые надо обновить, при изменении требований.

Инъекция зависимостей и тестирование

При тестировании модулей, которые зависят от других модулей, разработчики могут столкнуться с проблемой, когда необходимо создать экземпляры каждой из зависимостей.

Если модуль зависит от конкретной реализации другого, разработчикам придётся имитировать конкретную реализацию зависимости. Допустим, есть класс, который работает с хранилищем:

```
class StorageService {
  get(key: string): any {
    return JSON.parse(localStorage.getItem(key))
  }
}
```

Чтобы протестировать метод `get`, необходимо создать глобальный мок-объект `localStorage`. Такие объекты и переменные при тестировании могут привести к неправильной работе соседних тестов. (Например, если кто-то забыл сбросить `localStorage` после использования.)

Если же мы привяжем зависимость не через конкретный объект, а через интерфейс, то получим возможность подменять зависимости на лету. Этот паттерн называется инъекция зависимостей (Dependency injection, DI).

```
interface Storage {
  getItem(key: string): any
}

interface StorageDependencies {
  storage: Storage
}

class StorageService {
  storage: Storage

  // Указываем, какие зависимости следует использовать.
  // JSON тоже можно внедрить подобным образом,
  // но для простоты примера берём в расчёт только `localStorage`:

  constructor({ storage = localStorage }: StorageDependencies) {
    this.storage = storage
  }
```

```
  get(key: string): any {
    // Используем зависимость через интерфейс:
    return JSON.parse(this.storage.getItem(key))
  }
}
```

Теперь при тестировании мы можем указать мок-объект для `Storage` локально. При этом нам не потребуется эмулировать работу объекта `localStorage` полностью. Нам достаточно описать метод `getItem`, работу которого мы и проверим. Например, используя Jest:

```
describe('when called with a specific key', () => {
  it('should return the specified value', () => {
    const mock: Storage = {
      getItem: (key: string) => '42'
    }

    const service = new StorageService({storage: mock})
    expect(service.get('test key')).toEqual(42)
  })
})
```

Если нам важно проверить, вызвался ли правильный метод у зависимости, DI снова сделает решение задачи проще:

```
describe('when called with a specific key', () => {
  it('should call a specified method of the dependency object', () => {
    const mock: Storage = {
      getItem: jest.fn()
    }

    const service = new StorageService({storage: mock})
    service.get('test key')

    expect(mock.getItem).toHaveBeenCalled()
  })
})
```

Такой подход удобен при разработке через тестирование (TDD). Он позволяет продумать API модуля заранее и продумать организацию зависимостей модулей друг от друга.

Инъекция зависимостей и расширение функциональности

Теперь представим, что в приложении появляются два места, где используются разные хранилища: `localStorage` в одном и `verySophisticatedStorage` в другом.

Если наш класс зависел напрямую от `localStorage`, у нас проблемы. При добавлении нового хранилища, нам придётся проверять, с каким из хранилищ мы имеем дело. А если API хранилищ сильно отличается, то код метода `get` сильно разрастётся из-за проверок и адаптеров.

```
class StorageService {
  storageType: string
```

```
  constructor(storageType: string) {
    this.storageType = storageType
  }

  get(key: string): any {
    if (this.storageType === 'verySophisticatedStorage') {
      return verySophisticatedStorage.getByKey(key)
    }

    return JSON.parse(localStorage.getItem(key))
  }
}
```

С другой стороны, если мы зависим от интерфейса, то метод `get` и конструктор класса `StorageService` не изменятся. Разное API хранилищ мы приведём к одному виду через адаптеры (отдельные новые сущности), которые будут реализовывать интерфейс `Storage`.

```
interface Storage {
  getItem(key: string): any
}

// Добавляем адаптер для `verySophisticatedStorage`;
// он будет реализовывать интерфейс `Storage`,
// поэтому его можно будет передать как зависимость для `StorageService`:

class SophisticatedStorageAdapter implements Storage {
  getItem(key: string): any {
    return verySophisticatedStorage.getByKey(key)
  }
}

// То же для `localStorage`:

class LocalStorageAdapter implements Storage {
  getItem(key: string): any {
    return JSON.parse(localStorage.getItem(key))
  }
}

// Код класса `StorageService` не меняется!

class StorageService {
  storage: Storage

  constructor({ storage = localStorage }: StorageDependencies) {
    this.storage = storage
  }

  get(key: string): any {
    // Вся реализация методов обоих хранилищ скрыта за адаптерами:
```

```
    return this.storage.getItem(key)
  }
}

// Работает как со старым хранилищем:

const storageServiceWithLocalStorage = new StorageService({
  storage: new LocalStorageAdapter()
})

// ...Так и с новым:

const storageServiceWithSophisticatedStorage = new StorageService({
  storage: new SophisticatedStorageAdapter()
})
```

Таким образом новые бизнес-требования не затронут код уже созданного модуля `StorageService`, а будут внедрены через создание новых сущностей.

Материалы к разделу

- Инъекция зависимостей
- Мок-объект
- Разработка через тестирование
- Адаптер, шаблон
- Jest

Шаблоны проектирования и приёмы рефакторинга

Соблюдать принцип открытости-закрытости помогают несколько шаблонов проектирования и приёмов рефакторинга.

Абстрактная фабрика

Фабрика — это сущность, которая создаёт другие сущности по заданным правилам, например, создаёт экземпляры классов или объекты.

Абстрактная фабрика — это фабрика, которая создаёт фабрики.

Этот шаблон позволяет создавать сущности без привязки к конкретным классам — то есть без привязки к реализации. Так добавление новых типов сущностей изменяет минимальное количество модулей.

Рассмотрим применение абстрактной фабрики на том же примере с отчётом, что был у нас в разделе об SRP.

В этом примере работа для абстрактной фабрики — выбор класса, который будет заниматься форматированием.

```
// Часть с интерфейсом `Formatter` и классами,
// реализующими его, остаётся такой же.
// Добавляется новый интерфейс, описывающий фабрику фабрик:
```

```
interface FormatterFactory {
  createFormatter(data: ReportData): Formatter
}

// Метод `createFormatter` возвращает абстрактный интерфейс,
// поэтому обе фабрики ниже взаимозаменяемы:

class HtmlFormatterFactory implements FormatterFactory {
  createFormatter(data: ReportData) {
    return new HtmlFormatter(data)
  }
}

class TxtFormatterFactory implements FormatterFactory {
  createFormatter(data: ReportData) {
    return new TxtFormatter(data)
  }
}

// При конфигурации приложение выберет нужный
// тип фабрики и будет работать с ним.
// Коду приложения при этом будет не важно,
// с какой фабрикой он будет работать,
// потому что он будет зависеть от интерфейсов,
// а не от конкретных классов:

class AppConfigurator {
  reportType: ReportTypes

  constructor(reportType: ReportTypes) {
    this.reportType = reportType
  }

  configure(reportData: ReportData): FormatterFactory {
    if (this.reportType === ReportTypes.Html) {
      return new HtmlFormatterFactory(reportData)
    }
    else return new TxtFormatterFactory(reportData)
  }
}
```

Стратегия

В прошлом примере мы избавились от необходимости менять код форматеров при добавлении новых требований. Но вы могли заметить, что в методе `configure` класса `AppConfigurator` есть условие, которое проверяет тип формата для отчёта.

По-хорошему, подобные условия следует заменять на динамический выбор нужных сущностей. С этим может помочь ещё один шаблон — стратегия.

Этот шаблон позволяет менять настройки, конфигурацию или алгоритм в зависимости от ситуации и требований. В нашем случае стратегии мы отдадим выбор необходимой фабрики:

```
function formatterStrategy(reportType: ReportTypes) {
  const formatters = {
    [ReportTypes.Html]: HtmlFormatterFactory,
    [ReportTypes.Txt]: TxtFormatterFactory,
    default: TxtFormatterFactory
  }

  return formatters[reportType] || formatters.default
}
```

Теперь выбор фабрик стал динамическим, и при изменении требований нам потребуется только добавить новую сущность и обновить список фабрик.

Декоратор

Декоратор — это шаблон, который заключается в создании обёрток с дополнительной функциональностью для объектов. Такие обёртки позволяют не изменять сам объект, но при этом расширять его функциональность.

Отличие декоратора от наследования в возможности расширять функциональность динамически, без необходимости описывать каждый класс-наследник отдельно.

В примере мы создаём класс `BaseGreeting`, метод `greet` которого будет выводить строку с приветствием пользователя. Декоратор `GreetingWithUppercase` будет приводить строку с приветствием к верхнему регистру.

```
interface Greeting {
  username: string
  greet(): string
}

// Базовая функциональность описывается в классе,
// который будет обёрнут с помощью декораторов:

class BaseGreeting implements Greeting {
  username: string

  constructor(username: string) {
    this.username = username
  }

  greet(): string {
    return `Hello, ${this.username}!`
  }
}

// Здесь `decorated` — это объект,
// функциональность которого мы будем расширять:

interface GreetingDecorator {
  decorated: Greeting
```

```
  greet(): string
}

class GreetingWithUppercase implements GreetingDecorator {
  decorated: Greeting

  constructor(decorated: Greeting) {
    this.decorated = decorated
  }

  greet(): string {
    // 1. Используем базовое поведение:
    const baseGreeting = this.decorated.greet()

    // 2. Расширяем его:
    return baseGreeting.toUpperCase()
  }
}
```

Сама философия этого шаблона повторяет принцип открытости-закрытости — мы не меняем базовый класс, а добавляем сущности, которые содержат в себе изменения бизнес-требований.

Разница между декоратором и прокси в том, что прокси всегда предоставляет тот же интерфейс, в то время как декоратор может предоставлять расширенный интерфейс.

Наблюдатель

Наблюдатель — шаблон, который создаёт механизм подписки, когда некоторые сущности могут реагировать на поведение других.

В примере ниже у нас есть класс `SoftwareEngineerApplicant`, который реагирует на появление вакансий в экземпляре класса `HrAgency`.

```
// Интерфейс соискателя описывает его имя и желаемую позицию:

interface Applicant {
  name: string
  position: string
}

// Интерфейс наблюдателя описывает метод,
// который будет вызываться наблюдаемым объектом
// при наступлении события:

interface Observer {
  update(data: any): void
}

interface NewPositionObserver extends Applicant, Observer {
  update(position: string): void
}
```

```
class SoftwareEngineerApplicant implements NewPositionObserver {
  name: string
  position: string

  constructor(name: string, position: string) {
    this.name = name
    this.position = position
  }

  update(position: string) {
    if (position !== this.position) return
    console.log(`Hello! My name is ${this.name}, I would like to apply to a
${position} position`)
  }
}

// Базовый интерфейс наблюдаемого объекта:

interface Observable {
  subscribe(observer: Observer): void
  unsubscribe(observer: Observer): void
  notify(data: any): void
}

// Наблюдаемый объект хранит в себе список наблюдателей,
// которых будет уведомлять о наступлении события:

interface NewPositionObservable extends Observable {
  listeners: NewPositionObserver[]
  notify(position: string): void
}

class HrAgency implements NewPositionObservable {
  listeners: NewPositionObserver[] = []

  subscribe(applicant: NewPositionObserver): void {
    this.listeners.push(applicant)
  }

  unsubscribe(applicant: NewPositionObserver): void {
    this.listeners = this.listeners.filter(listener =>
      listener.name !== applicant.name)
  }

  // Уведомляем всех наблюдателей,
  // когда появляется новая вакансия:

  notify(position: string): void {
    this.listeners.forEach(listener => {
      listener.update(position)
    })
```

```
  }
}
```

Этот шаблон позволяет добавлять новых наблюдателей без необходимости менять код наблюдаемого объекта.

Замена прямого наследования на полиморфизм и композицию

Модули, зависящие от конкретных классов, связаны с ними слишком сильно, из-за чего изменение в одном модуле затронет изменение в другом.

Замена прямого наследования на полиморфизм или композицию — это приём рефакторинга, который ослабляет зацепление модулей через введение абстракций: интерфейсов, обёрток и т.д.

Примером может послужить пример из раздела «В идеальном мире», где мы вводили прослойку из интерфейса `Shape`. Когда класс `AreaCalculator` начинает зависеть от интерфейса, а не от конкретных классов, изменение в требованиях не затрагивает код класса `AreaCalculator`.

Материалы к разделу

- Composition over inheritance
- Фабрика
- Абстрактная фабрика
- Реализация абстрактной фабрики на TypeScript
- Стратегия
- Фабричный метод и применение стратегии
- Реализация стратегии на TypeScript
- Декоратор
- Decorator and Factory, Colorado Computer Science, PDF
- Реализация декоратора на TypeScript
- Наблюдатель
- Реализация наблюдателя на TypeScript
- Example of Just-in-time design: refactoring to OCP
- Essential Skills for the Agile Developer, Chapter 11: Refactor to the Open-Closed

Антипаттерны и запахи

Все паттерны, которые так или иначе теснее связывают сущности между собой, можно назвать антипаттернами.

Связывание через конкретные классы

Проблемы связывания через конкретные классы мы рассмотрели в примере из раздела «В идеальном мире». Класс `AreaCalculator` изначально зависел от конкретного класса `Rectangle`, из-за чего при изменении требований, приходилось обновлять код этого класса.

Когда мы ввели прослойку в виде интерфейса `Shape`, класс `AreaCalculator` начинал зависеть от интерфейса, а не от конкретных классов, изменение в требованиях перестало затрагивать код класса `AreaCalculator`.

Синглтон

Синглтон — это паттерн, при котором в приложении существует только один экземпляр какого-то класса. Существующий синглтон гарантирует, что все новые созданные объекты будут ссылаться на него.

Если синглтон содержит в себе общее состояние или глобальные переменные, либо сам является глобальной сущностью, то зацепление объектов с ним чрезмерно высокое.

Это может приводить к ситуациям, когда при изменении требований приходится менять и сам синглтон, и все объекты, которые с ним связаны.

Синглтон можно использовать и без высокого зацепления модулей (применяя, например, абстрактные интерфейсы), но это значительно усложняет структуру проекта.

Легковес

Легковес — шаблон, который уменьшает расход памяти, держа общее состояние объектов внутри себя, вместо хранения одинаковых данных в каждом объекте.

Его проблема с точки зрения OCP в том же высоком связывании модулей. Если сущность выступает в роли общего контекста для нескольких объектов, при изменении требований придётся менять код каждого модуля.

Как и в случае с синглтоном, с легковесом можно работать без высокого зацепления, но это также будет увеличивать сложность.

Материалы к разделу

- Синглтон
- Легковес
- Зацепление модулей
- STUPID → SOLID

Ограничения и подводные камни

У каждого принципа есть ограничения и область применения. Для OCP характерны следующие ограничения и подводные камни.

Система не может быть закрыта на 100%

Всегда есть изменения, которые невозможно внести, не изменив код какого-то модуля, поэтому при проектировании важен стратегический подход. Необходимо определить, от каких именно изменений и какие именно модули вы хотите закрыть.

Следует учитывать, что люди очень плохо умеют прогнозировать изменения. Даже имея достаточно знаний о системе и опыта, проектировщики не могут быть уверены, что предусмотрели все возможные варианты развития системы.

OCP предлагает подход Just-in-time design, при котором новые сущности добавляются в систему по мере необходимости, но не раньше. Это чем-то похоже на отказ от ранней оптимизации и раннего добавления абстракций.

Цель подхода в том, чтобы не создавать абстракции на пустом месте. Один из критериев хорошего дизайна — простота, поэтому использовать OCP следует всегда с оглядкой на то, насколько система получится простой в итоге.

Большое количество сущностей

Добавление функциональности менее рискованно, чем изменение существующей, но взамен мы рискуем увеличить количество сущностей. Бесконтрольное и бездумное следование OCP может приводить к ситуациям, когда интерфейсов станет слишком много, а функциональность — станет раздробленной.

Хороший дизайн системы — это в первую очередь простой дизайн. Чем меньше сущностей мы создаём для решения проблемы, тем выше вероятность, что дизайн хороший, поэтому следовать OCP необходимо с осторожностью.

Может быть не нужен для маленьких приложений

OCP нацелен на быстрое и дешёвое добавление функциональности и масштабирование системы. Он окупается, если система действительно большая, и проверять, как повлияло изменение кода, очень дорого или долго (или невозможно).

Если же приложение маленькое, то OCP может превратиться в принцип ради принципа — когда разработчики будут писать бойлерплейт-код для создания новых сущностей без видимой и ощутимой пользы.

OCP следует применять, если польза от него значительно выше, чем затраты от следования ему.

Материалы к разделу

- Just-in-time design
- Solve problem at hand
- Refactoring and OCP
- Дзен питона
- Бритва Оккама
- Not so SOLID principles

LSP

Введение

Одна из частых ошибок проектирования программных систем — это попытка полностью скопировать иерархию объектов из реального мира.

Моделируя систему, мы описываем *поведение* её компонентов, *отношения* их друг с другом, а не иерархию. Иерархия — удобный инструмент для моделирования, но иногда она приводит к неправильному описанию поведения.

Классический пример

Представим, что есть класс `Rectangle`, который описывает прямоугольник:

```
class Rectangle {
  width: number
  height: number

  constructor(width: number, height: number) {
    this.width = width
    this.height = height
  }

  setWidth(width: number) {
    this.width = width
  }

  setHeight(height: number) {
    this.height = height
  }

  areaOf(): number {
    return this.width * this.height
  }
}
```

Квадрат — тоже прямоугольник, мы можем использовать наследование, чтобы описать его:

```
class Square extends Rectangle {
  width: number
  height: number

  constructor(size: number) {
    super(size, size)
  }

  setWidth(width: number) {
    this.width = width
    this.height = width
```

```
  }

  setHeight(height: number) {
    this.width = height
    this.height = height
  }
}
```

Дальше в коде мы используем квадрат. Кажется, что всё в порядке:

```
declare const square: Square

square.setWidth(20) // Меняет ширину и высоту, всё верно.
square.setHeight(40) // Тоже меняет ширину и высоту, ок.
```

Но если мы используем класс `Rectangle` в качестве интерфейса, а работаем с конкретным классом `Square`, то могут возникнуть проблемы:

```
function testShapeSize(figure: Rectangle) {
  figure.setWidth(10)
  figure.setHeight(20)
  assert(figure.areaOf() === 200)

  // Условие не сработает, если `figure` —
  // экземпляр класса `Square`.
}
```

Разница между квадратом и прямоугольником в том, что у квадрата при изменении стороны меняются обе стороны, у прямоугольника — только одна, вторая остаётся неизменной.

Математически — да, квадрат всё ещё прямоугольник, но он *ведёт себя иначе*, чем прямоугольник.

Принцип подстановки Барбары Лисков

Принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP) решает эту проблему, вводя ограничения для иерархии объектов.

Звучит он так: функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Простыми словами — классы-наследники не должны противоречить базовому классу. Например, они не могут предоставлять интерфейс у́же базового. Поведение наследников должно быть ожидаемым для функций, которые используют базовый класс.

Немного удобнее думать об LSP в терминах «абстракция — реализация». Абстрактный класс или интерфейс играют роль базового типа, но вместе с этим — роль контракта на поведение.

```
abstract class Disposable {
  protected source: Optional<number>;
  constructor(fn: AnyFunction, delay: TimeIntervalMS) {}
```

```
  dispose(): void {}
}
```

Они гарантируют, что экземпляр любого конкретного класса будет содержать указанные поля и методы.

```
class Interval extends Disposable {
  constructor(fn: AnyFunction, delay: Milliseconds) {
    super(fn, delay);
    this.source = setInterval(fn, delay);
  }

  dispose = () => clearInterval(this.source);
}

class Timer extends Disposable {
  constructor(fn: AnyFunction, delay: Milliseconds) {
    super(fn, delay);
    this.source = setTimeout(fn, delay);
  }

  dispose = () => clearTimeout(this.source);
}
```

Это значит, что модуль, использующий этот *абстрактный класс или интерфейс* сможет работать с *любой* его реализацией. Например, функции `cleanup` будет неважно, экземпляр какого конкретного класса мы передадим:

```
function cleanup(disposable: Disposable): void {
  disposable.dispose();
}

const interval = new Interval(() => console.log('Hey!'), 1000);
const timer = new Timer(() => alert('Hey!'), 1000);

cleanup(interval);
cleanup(timer);
```

Принцип можно связать с понятиями ковариантности и контравариантности, а также с контрактным программированием. В частности полезны отношения предусловий и постусловий для базового и наследников:

- предусловия не могут быть усилены в подклассе;
- постусловия не могут быть ослаблены в подклассе.

Снова пример

В примере с `Rectangle` и `Square` последний ослабляет постусловие для методов `setWidth` и `setHeight`. Разберём, что это за постусловие.

Если мы работаем с методом `setHeight` класса `Rectangle`, то после вызова метода будем наблюдать ситуацию, когда:

```
const oldHeight = figure.height
figure.setWidth(newWidth)

assert((figure.width === newWidth) && (figure.height === oldHeight))
```

Но в случае с квадратом это не так. Постусловие — *свойства или состояние после выполнения метода* — ослабляется:

```
const oldHeight = figure.height
figure.setWidth(newWidth)

// Постусловие ослаблено,
// абстракция неправильная:

assert((figure.width === newWidth))
```

Из-за этого использовать `Rectangle` вместо `Square` без дополнительных проверок или изменения уже существующих компонентов невозможно.

Принцип подстановки Лисков требует использовать общий интерфейс для обоих классов и не наследовать `Square` от `Rectangle`.

Общий интерфейс должен быть таким, чтобы в классах, реализующих его, предусловия не были более сильными, а постусловия не были более слабыми.

Наследование и композиция

Следовать LSP проще, если не строить больших и глубоких иерархий. Обычно, для связи модулей *достаточно интерфейса и его реализаций*.

Длинные цепочки иерархий типов — хрупкие. Вместо иерархий типов лучше использовать композицию, чтобы собирать сущности из необходимой функциональности.

Например, наследование предполагает проектирование от общего к частному в виде иерархии:

```
Животные → Млекопитающие → Человек.
```

Но такие иерархии не всегда правильно отражают связи сущностей в проектируемой системе. Иногда иерархий вовсе может быть несколько одновременно, тогда наследование зайдёт в тупик — непонятно по какой иерархии наследоваться и как.

Композиция же подразумевает проектирование от частного к общему в виде совокупности нескольких обособленных наборов «фич»:

```
Человек = Скелет + Нервная система + Иммунная система + Сердечно-сосудистая система +
...
```

Таким образом мы не ограничиваем себя рамками одной иерархии, а значит нам необязательно подводить каждый аспект бизнес-логики под эту иерархию.

Более того, часто в отношениях сущностей, которые мы моделируем, в принципе нет никакой иерархии. В таком случае наследование сущностей наоборот лишь навредит.

Коротко

Принцип подстановки Барбары Лисков:

- помогает проектировать систему, опираясь на поведение модулей;
- вводит ограничения и правила наследования объектов, чтобы их потомки не противоречили базовому поведению;
- делает поведение модулей последовательным и предсказуемым;
- помогает избегать дублирования, выделять общую для нескольких модулей функциональность в общий интерфейс;
- позволяет выявлять при проектировании проблемные абстракции и скрытые связи между сущностями.

Материалы к разделу

- Принцип подстановки Барбары Лисков
- Контрактное программирование
- Ковариантность и контравариантность
- Принцип замещения Лисков
- What is an example of the Liskov Substitution Principle?
- The Liskov Substitution Principle, PDF
- Applying “Design by Contract”, Bertrand Meyer, PDF

В идеальном мире

Вернёмся к примеру с классами `Rectangle` и `Square` из введения.

Согласно LSP нам необходимо использовать общий интерфейс для обоих классов и не наследовать `Square` от `Rectangle`. Этот общий интерфейс должен быть таким, чтобы в классах, реализующих его, предусловия не были более сильными, а постусловия не были более слабыми.

У нас есть несколько способов решить эту проблему.

Абстрактный класс-родитель

Первый способ — переделать иерархию так, чтобы `Square` не наследовался от `Rectangle`. Мы можем ввести новый класс, чтобы и квадрат, и прямоугольник наследовались от него.

Создадим абстрактный класс `RightAngleShape`, чтобы описать фигуры с прямым углом:

```
abstract class RightAngleShape {
  // Используется для изменения ширины или высоты,
  // доступен только внутри класса и наследников:

  protected setSide(size: number, side?: 'width' | 'height'): void {}

  abstract areaOf(): number
}
```

Классы `Rectangle` и `Square` будут переопределять методы, поведение которых специфично для каждого из них:

```
class Square extends RightAngleShape {
  edge: number

  constructor(size: number) {
    super()
    this.edge = size
  }

  // Переопределяем изменение стороны квадрата...
  protected setSide(size: number): void {
    this.edge = size
  }

  setWidth(size: number): void {
    this.setSide(size)
  }

  // ...И вычисление площади:
  areaOf(): number {
    return this.edge ** 2
  }
}

class Rectangle extends RightAngleShape {
  width: number
  height: number

  constructor(width: number, height: number) {
    super()
    this.width = width
    this.height = height
  }

  // Переопределяем изменение ширины и высоты...
  protected setSide(size: number, side: 'width' | 'height'): void {
    this[side] = size
  }

  setWidth(size: number) {
    this.setSide(size, 'width')
  }

  setHeight(size: number) {
    this.setSide(size, 'height')
  }

  // ...И вычисление площади:
  areaOf(): number {
    return this.width * this.height
  }
}
```

Теперь поведение наследников не конфликтует с поведением базового класса. Это позволит использовать и `Rectangle`, и `Square` там, где объявлено использование `RightAngleShape`.

Интерфейс

Общий родительский класс — это только одно из решений. Мы помним, что наследование лучше заменить на более абстрактные вещи, например, на интерфейс. Второй способ заключается именно в использовании интерфейса.

Мы можем превратить родительский класс `RightAngleShape` в интерфейс `Shape` с описанным методом `areaOf`, а также описать интерфейсы для фигур, у которых есть ширина (`WidthfulShape`) и высота (`HeightfulShape`):

```
interface Shape {
  areaOf(): number
}

interface WidthfulShape {
  setWidth(size: number): void
}

interface HeightfulShape {
  setHeight(size: number): void
}
```

Классы `Rectangle` и `Square` тогда могут реализовать их так:

```
// Указываем, что необходимо реализовать в этом классе:

type SquareShape = Shape & WidthfulShape

class Square implements SquareShape {
  edge: number

  constructor(size: number) {
    this.edge = size
  }

  protected setSide(size: number): void {
    this.edge = size
  }

  // Указываем метод, меняющий ширину (описан в `WidthfulShape`)...
  setWidth(size: number) {
    this.setSide(size)
  }

  // ...И метод, который считает площадь (описан в `Shape`):
  areaOf(): number {
    return this.edge ** 2
  }
```

```
}


// Для прямоугольника, кроме площади и ширины,
// необходимо указать и высоту,
// поэтому добавляем интерфейс `HeightfulShape`:

type RectShape = Shape & WidthfulShape & HeightfulShape
type ShapeSide = 'width' | 'height'

class Rectangle implements RectShape {
  width: number
  height: number

  constructor(width: number, height: number) {
    this.width = width
    this.height = height
  }

  protected setSide(size: number, side: ShapeSide): void {
    this[side] = size
  }

  setWidth(size: number) {
    this.setSide(size, 'width')
  }

  // ...И реализуем метод, описанный в `HeightfulShape`:
  setHeight(size: number) {
    this.setSide(size, 'height')
  }

  areaOf(): number {
    return this.width * this.height
  }
}
```

Теперь с помощью интерфейсов мы можем композировать свойства, которые необходимо реализовать для сущностей. Мы автоматически соблюдаем принцип открытости-закрытости OCP, избегая прямого наследования и привязываясь к абстракциям, а не к конкретным классам.

Следуя же LSP, мы проектируем поведение сущностей так, чтобы оно не конфликтовало с базовой абстракцией. Это позволяет нам использовать любой из классов `Rectangle` или `Square` там, где заявлено использование как `Shape`, так и `WidthfulShape`.

Материалы к разделу

- The Liskov Substitution Principle, PDF
- SOLID Principles by Examples: Liskov Substitution Principle
- How does strengthening of preconditions and weakening of postconditions violate Liskov substitution principle?
- Applying “Design by Contract”, Bertrand Meyer, PDF

- The Liskov substitution principle, Adaptive Code Via Agile, PDF

В реальной жизни

Наследование предполагает иерархическую структуру сущностей, но с такими структурами есть проблемы, например — когда одна из сущностей не вписывается в эту иерархию.

Индикатор такой проблемы — проверки на принадлежность типу или классу перед выполнением какой-то операции или перед возвращением результата.

LSP помогает выявлять проблемные абстракции при проектировании и строить иерархию сущностей с учётом подобных проблем.

Как и принцип открытости-закрытости OCP, LSP подводит к выводу, что большие и сложные иерархии сущностей, основанные на наследовании, — это хрупкий и опасный инструмент, вместо которого лучше использовать композицию.

Иерархия пользователей

В одном из проектов стояла задача построить иерархию пользовательских ролей. Разработчики столкнулись с проблемой, когда один из типов пользователей не вписывался в существовавшую иерархию.

В проекте был класс `User`, который описывал сущность пользователя приложения. В нём были методы для работы с сессией, определением прав этого пользователя и обновлением профиля:

```
class User {
  constructor() {
    // ...
  }

  getSessionID(): ID {
    return this.sessID
  }

  hasAccess(action: Actions): boolean {
    // ...
    return access
  }

  updateProfile(data: Profile): CommandStatus {
    // ...
    return status
  }
}
```

Класс покрывал собой все роли пользователей, которые существовали в начале проекта: админ, руководитель группы пользователей, обычный пользователь.

В какой-то момент в приложении появился «гостевой режим». У гостей были ограниченные права, и не было профиля. Из-за отсутствия профиля в классе `Guest` метод `updateProfile` усиливал своё предусловие:

```
// Гости наследуются от пользователей:
class Guest extends User {
  constructor() {
    super()
  }

  hasAccess(action: Actions): boolean {
    // Описываем логику доступов для гостей:
    return access
  }

  updateProfile(data: Profile): CommandStatus {
    // А вот тут проблема: у гостей профиля нет,
    // из-за чего приходится выбрасывать исключение.
    // Гостевой режим как бы заставляет нас учитывать большее количество
    // обстоятельств, прежде чем выполнить обновление профиля:

    throw new Error(`Guests don't have profiles`)
  }
}
```

Применяем LSP

Попробуем решить проблему, применив LSP. Согласно принципу `Guest` должен быть заменяем на класс, от которого он наследуется, а приложение при этом не должно взрываться.

Введём общий интерфейс `User`, который будет содержать всё общее, что есть у гостей и пользователей.

```
interface User {
  getSessionID(): ID
}
```

Для описания доступов и работы с данными профиля создадим отдельные интерфейсы: `UserWithAccess` и `UserWithProfile`:

```
// Здесь всё, что относится к доступам:
interface UserWithAccess {
  hasAccess(action: Actions): boolean
}

// Здесь всё, что относится к профилю:
interface UserWithProfile {
  updateProfile(data: Profile): CommandStatus
}
```

Опишем базовый класс; от него будут наследоваться остальные классы гостей и пользователей:

```
class BaseUser implements User {
  constructor() {
```

```
    // ...
  }

  getSessionID(): ID {
    return this.sessID
  }
}

// У обычных пользователей добавляем методы
// для работы с профилем и для работы с доступами:

class RegularUser extends BaseUser implements UserWithAccess, UserWithProfile {
  constructor() {
    super()
  }

  hasAccess(action: Actions): boolean {
    // ...
    return access
  }

  updateProfile(data: Profile): CommandStatus {
    // ...
    return status
  }
}

// Для гостей же достаточно описать только доступы:

class Guest extends BaseUser implements UserWithAccess {
  constructor() {
    super()
  }

  hasAccess(action: Actions): boolean {
    // ...
    return access
  }
}
```

Теперь обновлять профиль мы можем только у сущностей, которые реализуют интерфейс `UserWithProfile`. Из-за этого проверять, является ли пользователь гостем, перед обновлением данных профиля не нужно, ведь гости не реализуют этот интерфейс, а значит такой функциональности у них нет.

Композиция или наследование

ООП — не про наследование и классы, а про отношение между сущностями и их поведение. В нём вполне успешно можно применять композицию — когда разные свойства объектов сочетаются в новом объекте.

При описании класса `RegularUser` в примере выше мы указали, что он реализует два интерфейса `UserWithAccess` и `UserWithProfile`. Каждый из интерфейсов отвечает за какую-то часть функциональности, которую мы сочетаем в `RegularUser` — это и есть композиция.

Преимущество композиции — в более высокой абстрактности, которая позволяет строить более гибкие отношения между сущностями.

React и JSX

Ещё один пример LSP — это React-компоненты, а точнее их реализация в JSX (JavaScript XML). Синтаксис JSX построен таким образом, что мы можем писать разметку компонентов в HTML-подобном виде:

```
interface ComponentProps {
  title: string
}

// Пример React-компонента:
const ExampleReactComponent: FunctionComponent<ComponentProps> = ({ title }) => (
  <div>
    <h1>{title}</h1>
    <OtherComponent />
  </div>
)
```

В примере выше можно заметить, что наряду с «обычными HTML-тегами» (`<div>` и `<h1>`) также рендерится и `<OtherComponent />` — другой React-компонент.

Возможность использовать компоненты точно так же, как «обычные теги», — это тоже реализация принципа подстановки Лисков. Мы можем заменить «обычный тег» на компонент, потому что и те и другие — это реализация `ReactElement`, который описывает, как именно они должны себя вести.

Материалы к разделу

- Liskov Substitution Principle, Hackernoon
- Liskov Substitution Principle, Maksim Ivanov
- How does strengthening of preconditions and weakening of postconditions violate Liskov substitution principle?
- Composition over inheritance
- Liskov Substitution Principle and the Composition Root - A Perspective
- Introducing JSX

Шаблоны проектирования и приёмы рефакторинга

Главная цель принципа подстановки Лисков — «исключить сюрпризы» в поведении объектов. Достигать этой цели помогают некоторые методы и шаблоны проектирования и приёмы рефакторинга.

Контрактное программирование

Контрактное программирование — это метод проектирования, при котором проектировщики чётко определяют и формализуют спецификации отношений между объектами.

Спецификации могут описывать интерфейсы методов, их пред- и постусловия, описание проверок и критерии соответствия. Такие спецификации называются контрактами.

В примере ниже интерфейс `Contract` описывает методы для проверки предусловия `require` и постусловия `ensure`. Класс `ContractAssert` реализует этот интерфейс, определяя, какие исключения следует сгенерировать при нарушении условий.

```
interface Contract {
  require(expression: boolean, msg?: string): void
  ensure(expression: boolean, msg?: string): void
}

class ContractAssert implements Contract {
  require(expression: boolean, msg?: string): void {
    if (!expression) throw new PreconditionException(msg)
  }

  ensure(expression: boolean, msg?: string): void {
    if (!expression) throw new PostconditionException(msg)
  }
}
```

Опишем исключения, наследуясь от стандартного `Error`. Класс `PreconditionException` отвечает за исключение при нарушении предусловия, `PostconditionException` — за нарушение постусловия.

```
class ContractException extends Error {
  constructor(msg?: string) {
    super(`contract error: ${msg}`)
  }
}

class PreconditionException extends ContractException {
  constructor(msg?: string) {
    super(`precondition failed, ${msg}`)
  }
}

class PostconditionException extends ContractException {
  constructor(msg?: string) {
    super(`postcondition failed, ${msg}`)
  }
}
```

Теперь если нам потребуется написать сумматор, который работает только с чётными числами, то мы можем проверять пред- и постусловия через контракт:

```
class EvenNumbersSummator {
  contract: Contract

  // Создаём контракт для проверки и записываем в `this.contract`:
```

```
  constructor(contract: Contract = new ContractAssert()) {
    this.contract = contract
  }

  add(a: number, b: number): number {
    // Перед работой метода проверяем все предусловия:
    this.contract.require(a % 2 === 0, 'first arg is not even')
    this.contract.require(b % 2 === 0, 'second arg is not even')

    const result = a + b

    // Перед тем, как вернуть результат проверяем постусловия:
    this.contract.ensure(result % 2 === 0, 'result is not even')
    this.contract.ensure(result === a + b, 'result is not equal to expected')
    return result
  }
}
```

Теперь метод не начнёт свою работу, если какое-то из предусловий будет нарушено, как и не вернёт результат, если будет нарушено постусловие.

Извлечение интерфейса, извлечение суперкласса

В прошлых разделах мы выделяли общий интерфейс для классов, которые не вписывались в существующую иерархию.

Для подобной работы подходят приёмы извлечения интерфейса и извлечения суперкласса. В примере с иерархией пользователей мы использовали выделение суперкласса.

Если разобрать изменения по шагам, то на первом шаге, чтобы не нарушать LSP, нам бы пришлось сделать `Guest` отдельным классом. Так мы бы избавились от усиления предусловий в методе `updateProfile`, но класс `Guest` начал бы дублировать функциональность, описанную в `User`:

```
// Класс не наследуется от `User`;
// метода `updateProfile` уже нет,
// но есть дублирование функциональности:

class Guest {
  constructor() {
    // ...
  }

  getSessionID(): ID {
    return this.sessID
  }

  hasAccess(action: Actions): boolean {
    // ...
    return access
  }
}
```

На втором шаге, чтобы избавиться от дублирования, мы бы применили выделение суперкласса. Класс `BaseUser` — и есть выделенный суперкласс.

Композиция, изменение модели наследования

Среди порождающих и структурных паттернов можно условно выделить группу таких, которые используют композицию свойств и методов. Это, например, стратегия и декоратор, которые мы рассмотрели в разделе OCP.

Кроме них «исключить сюрпризы» помогает изменение модели наследования. Не всегда иерархия объектов в программной системе должна копировать иерархию их в реальном мире. ООП — про отношение между сущностями и их поведение, поэтому модель наследования должна зависеть именно от поведения объектов.

Материалы к разделу

- Контрактное программирование
- SOLID: Liskov Substitution Principle
- The Contract Pattern, Michel de Champlain, PDF
- Design by Contract, JavaScript lib
- Извлечение интерфейса
- Извлечение суперкласса
- How to Correctly Model Inheritance
- Liskov substitution and abstract classes / strategy pattern
- Composition Root

Антипаттерны и запахи

Неправильная или неполная реализация некоторых шаблонов проектирования или приёмов, а также неправильная иерархия сущностей могут нарушить LSP.

Непредсказуемое изменение поведения

Допустим, мы делаем клон Медиума, где авторы будут публиковать статьи. Статья может находиться в разных состояниях, от которых зависит, что с ней можно делать. Например, удалённую статью нельзя удалить, а опубликованную — снова опубликовать.

Для подобной задачи подходит шаблон Состояние — он позволяет менять поведение объектов в зависимости от их внутреннего состояния. Если он реализован правильно и полно, то LSP он не нарушит. Однако реализация в примере ниже нарушает.

Допустим, статья описывается базовым классом `Article`:

```
enum ArticleStatus {
  Draft
  Published
  Deleted
}

class Article {
  status: ArticleStatus

  constructor() {/*...*/}
```

```
  edit() {/*...*/}
  delete() {/*...*/}
  restore() {/*...*/}
  unpublish() {/*...*/}

  publish(): void {
    this.status = ArticleStatus.Published
  }
}
```

Если опубликованная статья при попытке публикации выбрасывает исключение, *которое не было описано в базовом классе*, то мы нарушаем LSP:

```
class Published extends Article {
  constructor() {
    super({ status: ArticleStatus.Published })
  }

  publish(): void {
    // Упс!
    throw new Error('article is already published')
  }
}
```

Чтобы реализация шаблона не нарушала LSP, нам необходимо описать в базовом классе возможность выбросить исключение. Для этого мы введём метод, который будет определять, можно ли статью публиковать:

```
class ArticleException extends Error {/*...*/}

class Article {
  // ...

  protected canPublish(): boolean {
    return this.status === ArticleStatus.Draft
  }

  publish(): void {
    if (!this.canPublish()) throw new ArticleException()
    // ...
  }
}
```

Сейчас переопределение метода `publish` для опубликованной статьи не будет усиливать предусловия, поэтому это не нарушит LSP:

```
class ArticlePublishedException extends ArticleException {/*...*/}

class Published extends Article {
  // ...
```

```
  publish(): void {
    // `ArticlePublishedException` наследуется от `ArticleException`,
    // указанного в классе `Article`, поэтому здесь нарушения нет:

    throw new ArticlePublishedException()
  }
}
```

Интерфейс, которому нельзя доверять

Более тонкое нарушение LSP — это «пустая» реализация интерфейса.

Если опереться на пример выше, то пустой реализацией было бы описание метода `Publish` для опубликованной статьи таким образом:

```
class Published extends Article {
  // ...

  publish(): void {
    return
  }
}
```

Вроде всё хорошо: метод описывает правильное поведение, усиления предусловия нет. Но если посмотреть на ситуацию в терминах контрактного программирования, то метод `publish` должен менять состояние статьи с `Draft` на `Published`, чего не будет происходить:

```
class Article {
  // ...

  publish(): void {
    // Проверяем, что состояние позволяет публиковать статью
    // именно эта проверка в `Published` вызовет ошибку:

    this.contract.require(this.canPublish() === true)

    // ...

    // Проверяем, что состояние поменялось на `Published`:
    this.contract.ensure(this.status === ArticleStatus.Published)
  }
}
```

«Пустая» реализация интерфейса также нарушает принцип разделения интерфейса.

Материалы к разделу

- Состояние
- Does the state pattern violate LSP?
- Violating LSP

- Examples of LSP violation
- Common pitfalls and bad practices

Ограничения и подводные камни

Всем принципам приходится сталкиваться с реальностью, которая накладывает ограничения на их применение. Для LSP наиболее характерны дороговизна изменения модели иерархии, многословность контрактов, предел выделения суперкласса.

Изменение иерархии дорого

LSP помогает проектировать отношения между сущностями, учитывая возможные изменения в требованиях. Но вероятность спроектировать систему хорошо с первого раза — низкая.

Стремление следовать LSP может привести к изменению иерархии объектов в системе. Большие изменения в структурах — дороги, как по времени, так и по деньгам.

Также изменение структуры затрагивает большое количество компонентов. Если код плохо покрыт тестами, переписывание может привести к неработающему приложению.

Использование контрактов ресурсозатратно

Использование контрактов позволяет избежать усиления предусловий и ослабления постусловий. Однако, контракт увеличивает сложность проекта, а также увеличивает сроки разработки.

Покрытие методов контрактными спецификациями затратно по времени; контракты зачастую многословны и дублируют тесты. В некоторых случаях контракты снижают производительность кода.

Выделение суперкласса имеет предел

Выделение суперкласса позволяет избежать дублирования, но имеет предел вложенности. Невозможно выносить функциональность всё выше и выше без последствий — чрезмерное вынесение функциональности наверх может привести к появлению божественного объекта.

Также выделение суперкласса не гарантирует правильную иерархию объектов — может случиться, что новые бизнес-требования добавят сущность, которая не будет вписываться в уже изменённую иерархию.

Материалы к разделу

- How to Correctly Model Inheritance
- Контрактное программирование
- Извлечение суперкласса
- Божественный объект

ISP

Введение

Проблема банана и джунглей — одна из тех, за которые не любят ООП. Её суть простыми словами — когда при наследовании класс-потомок получает вместе с нужной функциональностью кучу неиспользуемой и ненужной.

Однако, проблема возникает не столько из-за ООП как такового, сколько из-за неправильной модели системы.

Принцип разделения интерфейса (Interface Segregation Principle, ISP) содержит правила и ограничения, которые помогают с этой проблемой справляться.

Принцип разделения интерфейса

Сущности не должны зависеть от интерфейсов, которые они не используют.

Когда принцип нарушается, модули подвержены *всем* изменениям в интерфейсах, от которых они зависят. Это приводит к высокой связанности модулей друг с другом.

ISP помогает проектировать интерфейсы так, чтобы изменения затрагивали только те модули, на функциональность которых они действительно влияют. Чаще всего это заставляет интерфейсы дробить (разделять).

Пример задачи

Допустим, требуется сделать программу, которая будет играть определённую ноту каждые несколько секунд. В наивном подходе с наследованием нам придётся построить нечто похожее на вот такую иерархию:

Проблема этой структуры в том, что мы представляем класс `SoundEmitter` потомком класса `TimeInterval`, хотя они друг с другом не связаны. Отсюда и возникает «необходимость» тащить ненужный код всем потомкам класса `TimeInterval`.

ISP предлагает два подхода к решению этой проблемы с помощью разделения интерфейсов: через делегирование и через множественное наследование.

Разделение через делегирование

Этот подход подразумевает использование шаблона проектирования под названием Адаптер.

Мы не будем связывать `SoundEmitter` и `TimeInterval` непосредственно друг с другом. Мы будем связывать их через дополнительный слой (адаптер), который будет транслировать сообщения от одной сущности другой.

Чуть подробнее об адаптере поговорим в разделе с шаблонами проектирования и приёмами рефакторинга.

Разделение через множественное наследование

Второй вариант предполагает, что `IntervalSoundEmitter` будет наследоваться одновременно от `TimeInterval` и от `SoundEmitter`. Это позволит отвязать родительские классы друг от друга и использовать в классе `IntervalSoundEmitter` только нужную функциональность.

Коротко

Принцип разделения интерфейса:

- помогает бороться с наследованием или реализацией ненужной функциональности;
- даёт возможность спроектировать модули так, чтобы их затрагивали изменения только тех интерфейсов, которые они действительно реализуют;
- снижает зацепление модулей;
- уничтожает наследование ради наследования, поощряет использование композиции;
- позволяет выявлять более высокие абстракции и находить неочевидные связи между сущностями.

Материалы к разделу

- Принцип разделения интерфейса
- Interface Segregation Principle, Object Mentor, Inc, PDF
- Goodbye, Object Oriented Programming
- Lamenting the Death of Object-Oriented Programming. Again?
- Why Object Oriented Matters
- Адаптер

В идеальном мире

В идеально спроектированной системе сущности зависят только от тех интерфейсов, функциональность которых реализуют. Чаще всего это приводит к дроблению интерфейсов на меньшие. Рассмотрим на примерах.

«Пустая» реализация

Допустим, у нас есть класс `Programmer`, который описывает программиста из офиса некоторой компании. Сотрудники пишут код и иногда едят пиццу, которую компания заказывает в офис.

```
interface Programmer {
  writeCode(): void
  eatPizza(slicesCount: number): void
}

class RegularProgrammer implements Programmer {
  constructor() {/*...*/}
  writeCode(): void {/*...*/}
  eatPizza(slicesCount: number): void {/*...*/}
}
```

Через какое-то время компания начала нанимать фрилансеров, которые работают удалённо и пиццу не едят. Если мы используем тот же интерфейс, то класс `Freelancer` должен будет реализовать метод `eatPizza`, хотя он ему и не нужен — это «пустая» реализация интерфейса.

```
class Freelancer implements Programmer {
  constructor() {/*...*/}
  writeCode(): void {/*...*/}
```

```
  eatPizza(slicesCount: number): void {
    // Здесь будет пусто —
    // это сигнал, что интерфейс надо дробить.
  }
}
```

Разделение интерфейса

Мы можем избежать проблемы из примера выше, если разделим интерфейс `Programmer`. Мы можем поделить его на две роли: `CodeProducer` и `PizzaConsumer`.

```
interface CodeProducer {
  writeCode(): void
}

interface PizzaConsumer {
  eatPizza(slicesCount: number): void
}
```

Теперь и `RegularProgrammer`, и `Freelancer` будут реализовывать только те интерфейсы, которые им действительно нужны:

```
class RegularProgrammer implements CodeProducer, PizzaConsumer {
  constructor() {/*...*/}
  writeCode(): void {/*...*/}
  eatPizza(slicesCount: number): void {/*...*/}
}

class Freelancer implements CodeProducer {
  constructor() {/*...*/}
  writeCode(): void {/*...*/}
  // Метод `eatPizza` уже не нужен.
}
```

Сравнение с SRP, влияние на LSP

ISP можно представлять как принцип единой ответственности (SRP) для интерфейсов. Дробление интерфейсов действительно заставляет делить ответственность между ними.

Если мы применяем ISP, получаем больше интерфейсов с меньшим количеством методов в каждом. Если мы применяем SRP, получаем больше модулей с меньшим количество методов в каждом. Применение обоих принципов разом заставляет делать контракты между модулями проще, что снижает вероятность нарушения принципа подстановки Лисков (LSP).

Материалы к разделу

- Interface Segregation Principle, Object Mentor, Inc, PDF
- Interface Segregation Principle
- The Single Responsibility Principle and the Interface Segregation Principle

В реальной жизни

ISP можно рассматривать как принцип единой ответственности (SRP) для интерфейсов. Его главная задача — помочь спроектировать интерфейсы так, чтобы в них не было ничего лишнего. Принцип помогает выявлять более высокие абстракции и находить неочевидные связи между сущностями.

Траты и доходы в Койне

Приложение Койн — это приложение для экономии, будущий преемник Тяжеловато и площадка для экспериментов.

В Койне есть возможность записывать траты и доходы. Разработчики использовали ISP, чтобы сделать работу с этими двумя сущностями более удобной.

Сперва в приложении были только траты, поэтому в истории хранились только объекты типа `Spend`.

```
interface Spend {
  amount: number
  date: Timestamp
  type: 'helpful' | 'harmful' | null
}

type History = Spend[]
```

Когда в приложении добавились доходы, в истории стали храниться не только траты. У трат и доходов были общие поля, поэтому было логично вынести их в общий интерфейс `Record`, а интерфейс траты расширить от него:

```
type RecordTypes = 'spend' | 'income'
type SpendTypes = 'helpful' | 'harmful' | null

// Общие поля описаны в общем интерфейсе.
// ISP помогает выявить скрытые связи
// между сущностями и зависимости в их поведении:

interface Record {
  amount: number
  date: Timestamp
  is: RecordTypes

  // Поля `type` здесь уже нет,
  // для доходов оно не нужно.
}

// Интерфейс траты расширяет интерфейс записи:

interface Spend extends Record {
  type: SpendTypes
}
```

Траты и доходы стали реализовать эти интерфейсы так, чтобы базовая запись описывала общее поведение, а трата и доход — только специфичное для них:

```
class RecordItem implements Record {
  amount: number
  date: Timestamp
  is: RecordTypes

  constructor(amount: number) {
    this.amount = amount
    this.date = Date.now()
  }
}

class SpendItem extends RecordItem implements Spend {
  type: SpendTypes

  constructor(amount: number, type: SpendTypes = null) {
    super(amount)
    this.is = 'spend'
    this.type = type
  }
}

class IncomeItem extends RecordItem {
  constructor(amount: number) {
    super(amount)
    this.is = 'income'
  }
}
```

Так функциональность не дублируется, но классы при этом не зависят от методов, которые им не нужны.

Уведомления в Задачнике

Задачник — это система управления задачами и учёта времени работы.

Для напоминания о подходящих сроках задачи в Задачнике используются уведомления. Уведомления бывают разных типов: push, SMS и почтовые.

Согласно ISP, общие для всех типов поля и методы разработчики хранят в общем интерфейсе `Message`:

```
interface Message {
  title: string
  body: string
  send(to: string[]): void
}
```

Детали же — описывают конкретно под каждый тип уведомлений:

```
interface SmsMessage extends Message {
  smsService: SmsService
  // ...
}

interface PushMessage extends Message {
  pushService: PushService
  // ...
}

interface EmailMessage extends Message {
  emailService: EmailService
  // ...
}
```

Шаблоны проектирования и приёмы рефакторинга

Следовать принципу разделения интерфейса помогают такие шаблоны проектирования как Адаптер, а также приёмы выделения интерфейса и множественного наследования.

Адаптер

Адаптер — шаблон проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

С точки зрения ISP этот шаблон помогает держать интерфейсы чистыми и понятными, а при необходимости совместить несовместимые модули через специальную прослойку (адаптер).

Приложение Курсовик показывает курс доллара к рублю. Оно берёт данные курсов с сайта Центрального банка России. Центробанк отдаёт их в формате XML, а Курсовик работает с JSON.

Адаптер помогает «подружить» модули, которые работают с XML, и модули, которые работают с JSON.

```
const ApiClient = {
  async getXml(url: string): Promise<XmlString> {
    const response = await fetch(url)
    const data = await response.text()
    return data
  }
}

const xmlJsonAdapter = (xml: XmlString): JsonString => {
  // Конвертируем xml в json:
  return json
}

const parseCourse = (data: JsonString): CourseDict => {
  // ...
  return course
```

```
}

(async () => {
  const data = await ApiClient.getXml('api_endpoint')
  const course = parseCourse(xmlJsonAdapter(data))
})()
```

Также адаптер помогает справляться со «сломанным» API от бекенда и преобразованием одних структур данных в другие.

Из минусов можно назвать:

- добавление ещё одной абстракции в кодовую базу проекта;
- при создании нового адаптера нужно найти все места, где требуется его использовать.

Выделение интерфейса

Выделение интерфейса — это приём, при котором одинаковые методы и поля выносят в отдельный интерфейс.

В качестве примера можно вернуться к Койну из прошлого раздела. Интерфейс `Record` — это выделенный общий интерфейс, который включает в себя общие для траты и дохода поля.

Выделение интерфейса тесно связано не только с ISP, но и с LSP. Например, оно используется при поиске корня композиции и как вспомогательный инструмент для выделения суперкласса.

Множественное наследование

Множественное наследование используется, например, чтобы реализовать функциональность нескольких интерфейсов:

```
class Horse implements Animal, Transport {/*...*/}
```

В TypeScript такое наследование реализуется через миксины.

До этих пор в книге для простоты повествования мы пропускали специальную функцию `applyMixins`, которая копирует функциональность из родительских классов:

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach(baseCtor => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
      Object.defineProperty(derivedCtor.prototype, name,
Object.getOwnPropertyDescriptor(baseCtor.prototype, name))
    })
  })
}
```

Чтобы пример с классом `Horse` выше сработал, нам необходимо использовать `applyMixins` следующим образом:

```
applyMixins(Horse, [Animal, Transport])
```

Тогда множественное наследование будет работать, как мы ожидаем.

Материалы к разделу

- Адаптер
- Лекарство от сломанной обратной совместимости
- Реализация адаптера на TypeScript
- Выделение интерфейса
- Liskov Substitution Principle and the Composition Root
- Множественное наследование
- Mixins in TypeScript

Антипаттерны и запахи

Принцип разделения интерфейса концептуально похож на принцип единственной ответственности и решает схожие проблемы, только относящиеся к более высокому уровню абстракции.

Два самых распространённых «запаха», намекающих, что нарушен ISP, — это грязный интерфейс и «пустая» реализация.

Грязный интерфейс

Проблема грязного интерфейса возникает, когда интерфейс содержит в себе слишком много методов и полей. Как и большие модули в SRP, грязный интерфейс в ISP приводит к плохой читаемости и дорогой поддержке.

Рассмотрим проблему на примере. Допустим, у нас есть калькулятор, который умеет складывать, вычитать, умножать, делить и брать квадратные корни.

```
interface Calculator {
  add(a: number, b: number): number
  subtract(a: number, b: number): number
  multiply(a: number, b: number): number
  divide(a: number, b: number): number
  sqrt(a: number): number
}

class BasicCalculator implements Calculator {
  // ...
}
```

При расширении функциональности интерфейс может стать слишком большим:

```
interface Calculator {
  add(a: number, b: number): number
  subtract(a: number, b: number): number
  multiply(a: number, b: number): number
  divide(a: number, b: number): number
  sqrt(a: number): number
  power(base: number, power: number): number
  sin(x: number): number
  cos(x: number): number
```

```
  tan(x: number): number
  log(x: number): number
  log10(x: number): number
  // ...
}
```

Класс, реализующий такой интерфейс будет неоправданно большим, из-за чего сложность понимания кода резко вырастет.

Кроме высокой сложности грязные интерфейсы приводят и ещё к одной проблеме, которую мы упоминали в разделе с примерами из идеального мира — пустой реализации.

«Пустая» реализация

«Пустая» реализация интерфейса возникает, когда появляется модуль, которому не нужны *все* методы из реализуемого интерфейса.

Попробуем создать арифметический калькулятор. Ему не нужны тригонометрические функции и логарифмы, но из-за того, что они описаны в интерфейсе, нам придётся поставить заглушки на эти методы:

```
class ArithmeticCalculator implements Calculator {
  // Функциональность методов, которые нужны,
  // описываем полностью:

  add(a: number, b: number): number {
    return a + b
  }

  subtract(a: number, b: number): number {
    return a - b
  }

  multiply(a: number, b: number): number {
    return a * b
  }

  divide(a: number, b: number): number {
    if (b === 0) {/*...*/}
    return a / b
  }

  // Остальные методы не нужны,
  // но интерфейс `Calculator` требует их реализации,
  // приходится писать заглушки:

  sqrt(a: number): number
  power(base: number, power: number): number
  sin(x: number): number { return 0 }
  cos(x: number): number { return 0 }
  tan(x: number): number { return 0 }
  log(x: number): number { return 0 }
```

```
  log10(x: number): number { return 0 }
  // ...
}
```

ISP решает

ISP помогает проектировать интерфейсы, избегая проблем выше. На примере калькулятора мы бы могли выделить несколько интерфейсов с соответствующими ролями:

```
interface ArithmeticCalculator {
  add(a: number, b: number): number
  subtract(a: number, b: number): number
  multiply(a: number, b: number): number
  divide(a: number, b: number): number
}

interface ExponentCalculator {
  sqrt(a: number): number
  power(base: number, power: number): number
}

interface TrigonometricCalculator {
  sin(x: number): number
  cos(x: number): number
  tan(x: number): number
}

interface LogarithmicCalculator {
  log(x: number): number
  log10(x: number): number
}
```

Тогда арифметическому калькулятору достаточно было бы реализовать интерфейс `ArithmeticCalculator`. Более навороченный инженерный калькулятор смог бы реализовать несколько интерфейсов одновременно с помощью множественного наследования.

Материалы к разделу

- Avoiding Interface Pollution with the ISP
- RoleInterface, Martin Fowler
- Принцип разделения интерфейса
- Can anyone provide an example of the LSP?
- Шаблон Null-object
- Does Null-object Pattern Break ISP?

Ограничения и подводные камни

Как и в случае с SRP, проблемы с разделением интерфейсов возникают из трудностей проектирования и прогнозирования.

Слепое следование опасно

Слепое следование ISP грозит дроблению *вообще всех* интерфейсов на атомарные с одним методом или полем. Это похоже на атомарный CSS, когда каждое свойство описывается отдельным классом.

Но суть принципа не в том, чтобы раздробить интерфейсы, а в том, чтобы выделить *минимально необходимое количество методов*, чтобы реализующие его модули не зависели от ненужной функциональности.

Как понять, что интерфейс стал грязным

Во многих предметных областях сущности могут обладать очень большим количеством свойств и методов, из-за чего разделять интерфейсы может быть трудно. Разработчики могут столкнуться с ситуацией, когда не будет очевидно, в какой момент интерфейс всё ещё подчиняется ISP, а в какой — уже нет.

Конфликты ролей и иерархий

Интерфейсы можно условно поделить на роли: тип, поведение, ожидание (например в контракте) и др.

Иерархия ролей может конфликтовать с иерархией сущностей и модулей, которые реализуют интерфейсы. Это делает структуру системы сложной для понимания и может вводить в заблуждение при чтении кода.

Материалы к разделу

- Атомарный CSS
- The 6 roles of the interface
- Контрактное программирование
- Goodbye, Object Oriented Programming

DIP

Введение

Программные системы состоят из модулей, которые мы можем условно поделить на низкоуровневые и высокоуровневые.

Низкоуровневые содержат утилитарную функциональность: обращение к базе данных, запросы к серверу, рендеринг DOM-элементов на странице.

Высокоуровневые содержат сложную, более абстрактную бизнес-логику. Они достаточно абстрактны, чтобы их можно было переиспользовать в разных проектах: авторизация пользователей, валидация форм, отправка уведомлений.

В устойчивых системах высокоуровневые модули, как правило, не требуют обновления при изменении низкоуровневых. Добиться подобной устойчивости помогает принцип инверсии зависимостей.

Принцип инверсии зависимостей

Принцип инверсии зависимостей (Dependency Inversion Principle, DIP) предполагает, что:

- Высокоуровневые модули не должны зависеть от низкоуровневых; оба типа должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей, детали должны зависеть от абстракций.

Таким образом DIP помогает снизить зацепление модулей (coupling).

Когда модули жёстко зацеплены, они слишком много знают друг о друге и не функционируют по отдельности. В такой ситуации изменения в одном будут требовать изменений в других — что нарушает OCP.

Зацепление и связность

Зацепление (coupling) не стоит путать со *связностью* (cohesion).

Зацепление — степень взаимозависимости разных модулей. Чем выше зацепление, тем более хрупкой получается система, и тем сложнее вносить изменения.

Связность — степень, в которой задачи некоторого модуля связаны друг с другом. Чем выше связность, тем строже модули следуют SRP, тем выше сфокусирован модуль на конкретной задаче.

Абстракции

Согласно принципу модули должны зависеть от других модулей не напрямую, а от абстракций.

В примере с регулятором температуры на схеме ниже структура системы нарушает DIP. Модули зависят напрямую от других модулей, это увеличивает зацепление.

Исправленный вариант вводит прослойку между сущностями в виде абстракций — интерфейсов.

Модулям теперь не обязательно работать с конкретными модулями, они могут работать с любой сущностью, которая реализует указанный интерфейс. Так снижается зацепление.

DIP и тестируемость

При тестировании модуля, который зависит от других модулей, нам нужно либо создавать экземпляр каждой зависимости, либо создать заглушки.

DIP упрощает тестирование системы. Если модули зависят от интерфейсов, нам достаточно создать заглушку, реализующую этот интерфейс.

Коротко

Принцип инверсии зависимостей:

- вводит правила и ограничения для зависимости одних модулей от других;
- снижает зацепление модулей;
- делает тестирование модулей проще;
- позволяет проектировать систему так, чтобы модули были заменяемы на другие.

Материалы к разделу

- Принцип инверсии зависимостей, Википедия
- Принцип инверсии зависимостей, Александр Бындю
- Принцип инверсии зависимостей, MakeDev
- Зацепление модулей
- Связность модулей
- General Responsibility Assignment Software Patterns, GRASP
- Inversion of Control Containers and the Dependency Injection pattern
- Пример с терморегулятором

В идеальном мире

В идеально спроектированной системе модули зависят только от абстракций, зацепление — минимально, а связность — максимальна.

Аутентификация пользователей

Рассмотрим в качестве примера модуль аутентификации пользователей.

```
class MySqlConnection {/*...*/}

class Auth {
  connection: MySqlConnection

  constructor(connection: MySqlConnection) {
    this.connection = connection
  }

  async authenticate(login: string, password: string): Promise<AuthResult> {/*...*/}
}
```

В примере выше модуль `Auth` — высокоуровневый, а `MySqlConnection` — низкоуровневый. Пример нарушает DIP, потому что `Auth` зависит напрямую от `MySqlConnection`.

Если мы сменим базу данных, то нам придётся менять и код модуля `Auth`. По-хорошему, `Auth` не должен ничего знать о базе данных, которую мы используем. Ему достаточно знать, что есть какая-то база, к которой можно достучаться через определённые методы — это работа интерфейса.

```
interface DataBaseConnection {
  connect(host: string, user: string, password: string): void
}

class MySqlConnection implements DataBaseConnection {
  constructor() {/*...*/}
  connect(host: string, user: string, password: string): void {/*...*/}
}
```

Теперь мы можем отвязать `Auth` от конкретной базы данных, указав в зависимости интерфейс.

```
class Auth {
  // Тип зависимости поменялся:
  connection: DataBaseConnection

  constructor(connection: DataBaseConnection) {
    this.connection = connection
  }

  authenticate(login: string, password: string) {/*...*/}
}
```

OCP и LSP автоматом

Исправленный вариант автоматически удовлетворяет двум другим принципам: открытости-закрытости (OCP) и подстановки Лисков (LSP).

Мы можем заменить одну базу данных на другую, если она реализует интерфейс `DataBaseConnection`, и приложение не сломается, как этого требует LSP. При изменении базы нам не придётся изменять код модуля `Auth` — так мы удовлетворяем OCP.

Материалы к разделу

- Принцип инверсии зависимостей, MakeDev
- Dependency Inversion Principle, oodesign
- OCP vs DIP

В реальной жизни

Для большего контроля над зависимостями и упрощения тестирования DIP предлагает использовать инверсию управления (Inversion of Control, IoC) и инъекцию зависимостей (Dependency Injection, DI).

DIP, DI и тестирование

Писать тесты без DIP возможно, но следуя принципу это делать проще. Если модуль зависит от абстракции, то его зависимость легче подменить фейковым объектом.

Вернёмся к примеру с аутентификацией. Для того, чтобы протестировать реализацию, которая не следует DIP, нам придётся создать экземпляр класса `MySqlConnection`:

```
describe('Auth', () => {
  let connection: MySqlConnection
  let auth: Auth

  beforeEach(() => {
    connection = new MySqlConnection(/*...*/)
    auth = new Auth(connection)
  })

  it('should successfully authenticate user', async () => {
    const result: AuthResult = await auth.authenticate('Alex', '123')
    expect(result.status).toEqual(200)
  })
})
```

Проблема — в производительности. Хороший модульный тест должен быть быстрым, а создавать экземпляр настоящего объекта на каждый тест — ресурсозатратно.

Кроме того, в плохой реализации класс `MySqlConnection` сам может зависеть от других модулей, экземпляры которых тоже будут создаваться. Всё это сильно тормозит тесты.

Теперь попробуем протестировать реализацию, которая следует DIP:

```
class DbMock implements DataBaseConnection {
  connect(host: string, user: string, password: string): void {
    // Здесь дешёвые операции-заглушки,
    // подключаться к базе совсем не обязательно.
  }

  // Тут реализация только тех методов,
  // которые понадобятся для тестов.
}

describe('Auth', () => {
  let connection: DataBaseConnection
  let auth: Auth

  beforeEach(() => {
    connection = new DbMock(/*...*/)
    auth = new Auth(connection)
  })

  // ...
})
```

Класс `DbMock` реализует интерфейс `DataBaseConnection`. Мы можем спроектировать его максимально простым и лёгким, это ускорит работу теста.

Инверсия управления

DI — это частный случай инверсии управления. При этом подходе контроль за выполнением программы отдаётся фреймворку, который знает, в какой момент и какую функцию надо вызвать. Цель IoC — сделать систему расширяемой.

Inversify предлагает решение для IoC на TypeScript. Inversify предоставляет API для создания контейнеров с указанием зависимостей, которые потом подставляются автоматически.

Материалы к разделу

- Dependency Injection
- Inversion of Control
- What's the difference between faking, mocking, and stubbing?
- Mock Objects, Design and DIP
- Пример разработки Крестиков-ноликов по TDD
- On DI, Loose Coupling and Unit Tests
- Inversify.io
- Dependency Inversion with Inversify.io

Шаблоны проектирования и приёмы рефакторинга

Следовать принципу инверсии зависимостей помогают инъекция зависимостей, наблюдатель и шаблонный метод.

Инъекция зависимостей

Внедрять зависимости можно тремя способами: через конструктор, через сеттеры и интерфейсно.

Инъекция через конструктор

Самый распространённый вид инъекции — через конструктор. При создании класса в конструкторе мы перечисляем все зависимости, которые требуются для создания экземпляра:

```
class Room {
  private chair: Chair
  private couch: Couch
  private table: Table

  constructor(chair: Chair, couch: Couch, table: Table) {
    this.chair = chair
    this.couch = couch
    this.table = table
  }
}

// Или используя короткий вариант записи:

class Room {
  constructor(
    private chair: Chair,
    private couch: Couch,
    private table: Table
```

```
  ) {}
}
```

Непосредственно внедрением обычно занимается специальная сущность — DI-контейнер. Контейнеры автоматизируют создание экземпляров нужных классов и помогают в управлении их жизненным циклом.

В языках со статической типизацией DI опирается на *типы* зависимостей, объявленных в конструкторе. Используя эти типы и конфигурацию приложения, контейнер выбирает конкретные классы, которые будет внедрять:

```
const container = new DIContainer();

// При регистрации зависимостей мы будто объявляем «соответствие»
// между абстрактным типом или интерфейсом и конкретной реализацией:

container.registerSingleton<Chair, WoodenChair>();
container.registerSingleton<Couch, LeatherCouch>();
container.registerSingleton<Table, DiningTable>();

// Мы будто говорим:
// — Контейнер, если увидишь в коде тип `Chair`,
//   замени его на экземпляр класса `WoodenChair`.
```

Это, например, помогает подменять зависимости во время тестирования, заменив регистрацию:

```
// Во время тестирования мы укажем другие классы,
// которые будут реализовывать те же интерфейсы,
// но, например, будут представлять собой моки или стабы:

container.registerSingleton<Chair, TestChair>();
container.registerSingleton<Couch, TestCouch>();
container.registerSingleton<Table, TestTable>();
```

В TypeScript также существуют DI-контейнеры, которые работают на декораторах, но подобное внедрение выглядит не так чисто и требует изменений непосредственно кода. Внедрение же, основанное на типах, помогает отделить «инфраструктурный» код от кода приложения.

Инъекция через конструктор отлично подходит, когда все или большая часть зависимостей необходимы для работы класса. Если же зависимости опциональны или нам нужно менять их в процессе работы приложения, мы можем воспользоваться инъекцией через сеттер.

Инъекция через сеттер

При внедрении через сеттер каждая зависимость указывается в поле, которое можно изменить через `set`. С таким видом внедрения мы не можем гарантировать наличие всех зависимостей, поэтому они должны быть опциональными, а их отсутствие не должно нарушать работу класса.

Например, наличие в комнате мебели не обязательно, комната сможет «работать» и без неё, потому зависимости могут быть опциональными:

```
class Room {
  public chair?: Chair
  public couch?: Couch
  public table?: Table
}
```

Мы можем создать комнату без мебели, а потом обставлять её по мере необходимости:

```
const livingRoom = new Room();

livingRoom.couch = new LeatherCouch();
livingRoom.table = new OakTable();
```

Это же позволяет заменять зависимости во время работы, что может быть полезно при использовании паттерна «Стратегия»:

```
livingRoom.couch = new LeatherCouch();

// ...Спустя какое-то время диван можно заменить:

livingRoom.couch = new FabricCouch();
```

Проблема этого подхода в том, что поля с зависимостями становятся `public`, а это не всегда приемлемо. Если объявлять поля публичными нам не подходит, мы можем воспользоваться внедрением с помощью интерфейса.

Инъекция с помощью интерфейса

Внедрение с помощью интерфейса похоже на предыдущий подход, только в нём используются не сеттеры, а отдельные методы-инжекторы. Их мы описываем в интерфейсе `RoomBuilder`, который реализует класс `Room`:

```
interface RoomBuilder {
  injectChair(dep: Chair): void
  injectCouch(dep: Couch): void
  injectTable(dep: Table): void
}

class Room implements RoomBuilder {
  private chair?: Chair
  private couch?: Couch
  private table?: Table

  injectChair(chair: Chair) {
    this.chair = chair
  }

  injectCouch(couch: Couch) {
    this.couch = couch
  }
```

```
  injectTable(table: Table) {
    this.table = table
  }
}
```

Часто такое внедрение используют вместе с паттерном «Фабрика» или «Строитель» для более явного построения экземпляра. Например, внутри функции `roomFactory` мы создаём экземпляр класса `Room`, а потом вызываем инжекторы, передавая в аргументах нужные зависимости:

```
function roomFactory(): Room {
  const room = new Room()

  room.injectChair(new WoodenChair())
  room.injectTable(new WoodenTable())

  return room
}
```

Наблюдатель

Наблюдатель — шаблон, который создаёт механизм подписки, когда некоторые сущности могут реагировать на поведение других.

Наблюдатель инвертирует контроль за выполнением программы схожим образом, как это делают обработчики событий в GUI. Обработчики событий вызываются в момент пользовательского события ввода: щелчок мыши, нажатие клавиши; наблюдатель — реагирует на изменение состояния наблюдаемого объекта.

В примере из раздела об OCP класс `SoftwareEngineerApplicant` следит за появлением новой вакансии у `HrAgency`. Метод `update` решает, как обработать изменение состояния.

Взаимодействие классов `SoftwareEngineerApplicant` и `HrAgency` «становится фреймворком», который следит за изменениями и вызывает нужные методы.

Шаблонный метод

Шаблонный метод — это шаблон, который определяет скелет алгоритма, а некоторые шаги даёт реализовывать подклассам. Так подклассы могут переопределять части алгоритма, не меняя общей структуры.

В примере ниже шаблонный метод `brewBeverage` задаёт каркас алгоритма приготовления напитка.

```
abstract class BeverageMachine {
  public brewBeverage(): Beverage {
    this.turnOn()
    this.prepareIngredients()
    this.prepareContainer()
    this.brew()
    this.hook()
  }
```

```
  // Базовые операции имеют реализацию:

  public turnOn(): void {
    this.on = true
  }

  // Специфичные для каждого подкласса операции
  // будут переопределяться потомками:

  abstract public prepareIngredients(): void
  abstract public prepareContainer(): void
  abstract public brew(): void

  // Хуки предоставляют дополнительные точки расширения
  // в некоторых критических местах алгоритма.
  // Их переопределять не обязательно,
  // так как есть пустая реализация по умолчанию:

  public hook(): void {}
}
```

Конкретные классы реализуют абстрактные методы базового. Они также могут переопределить и некоторые методы по умолчанию. Как правило, конкретные переопределяют только часть функциональности.

```
class CoffeeMachine extends BeverageMachine {
  abstract public prepareIngredients(): void {
    this.grindBeans()
    this.heatMilk()
  }

  abstract public prepareContainer(): void {
    this.getNewCup()
  }

  abstract public brew(): void {
    this.pourEspresso()
    this.pourMilk()
  }

  // ...
}
```

В стандартной модели наследования потомки вызывают методы базового класса. Здесь же наоборот — методы, реализованные в конкретных классах, вызываются в базовом через шаблонный метод.

Преимущество такого подхода в повторном использовании алгоритма с различными вариациями. Опасность шаблона — в случайном нарушении LSP при изменении функциональности подкласса.

Материалы к разделу

- Наблюдатель

- Стратегия
- Реализация наблюдателя на TypeScript
- Шаблонный метод
- Реализация шаблонного метода на TypeScript
- Инъекция зависимостей
- Three Design Patterns That Use IoC
- Dependency Injection and its variations
- Внедрение зависимостей с TypeScript на практике
- What is a DI Container?

Антипаттерны и запахи

Основная проблема антипаттернов и запахов ниже — в сокрытии зависимостей и увеличении зацепления модулей друг с другом.

Контрол-фрик

Контрол-фрик — это запах, при котором зависимости явно создаются в конструкторе с помощью `new`.

```
class Auth {
  // Обратите внимание на отсутствие аргументов у конструктора
  // и ключевое слово `new` внутри:

  constructor() {
    this.connection = new MySqlConnection()
  }
}
```

Класс `Auth` становится жёстко зацепленным с `MySqlConnection`, то есть *не может работать без него*. Это нарушает OCP, LSP и ISP.

Также чтобы протестировать такой класс, придётся сделать *глобальный* мок для `MySqlConnection`. Это ресурсозатратно и засоряет глобальную область видимости.

Локатор служб

Локатор служб — шаблон, который позволяет подключать зависимости в момент, когда они становятся нужны.

Мы называем его антипаттерном с оговоркой, что есть ситуации, когда он нужен — например, при работе с легаси. Но при создании новой системы лучше использовать DI.

Самый главный минус этого шаблона в том, что он нарушает инкапсуляцию, скрывая зависимости.

Понять код и предугадать его поведение становится трудно — ведь, чтобы узнать все зависимости необходимо изучить исходники. IDE уже не сможет подсказать, какие зависимости необходимы для создания класса.

При использовании DI такой проблемы нет — у нас просто не получится создать экземпляр класса,не передав все необходимые зависимости.

Материалы к разделу

- Dependency Injection: антипаттерны
- Локатор служб
- Is Service Locator an Antipattern?
- Service Locator is an Antipattern, Mark Seemann

Ограничения и подводные камни

Принцип инверсии зависимостей может приводить к ситуациям, когда поток управления программы задан неявно, а явное — лучше неявного.

Неграмотное использование

При неграмотном и бесконтрольном использовании принципа и IoC-контейнеров в частности логика взаимодействия компонентов может оказаться размазанной по разным частям приложения. Это мешает восприятию программы, снижает тестируемость и увеличивает стоимость внесения изменений.

Кроме того, не все зависимости стоят того, чтобы их инвертировать. Отличить такие от остальных бывает трудно из-за, например, специфики предметной области.

Проблемы с внедрением зависимостей

С инъекцией зависимостей тоже могут возникать проблемы.

При инъекции через конструктор большое количество зависимостей сильно увеличивает конструктор. А при неграмотном использовании DI могут появиться циклические зависимости.

В обоих случаях стоит посмотреть в сторону SRP. Скорее всего, класс делает слишком много, и его надо разбивать на более мелкие.

Материалы к разделу

- Дзен Питона
- Принцип инверсии зависимостей
- Разбираемся с SOLID: Инверсия зависимостей
- Dependency Injection and its variations
- Dependency Hell

Заключение

В этой книге мы рассмотрели 5 принципов объектно-ориентированного дизайна, их пользу в проектировании систем и ограничения. Поговорили:

- о принципе единственной ответственности;
- открытости и закрытости;
- подстановки Барбары Лисков;
- разделения интерфейса;
- инверсии зависимостей.

Обсудили шаблоны проектирования, которые помогают или мешают следовать этим принципам. Изучили запахи кода, которые сигнализируют о проблемах и несоблюдении принципов.

Что дальше?

Проектирование систем — штука междисциплинарная. Чтобы спроектировать устойчивую систему, надо не только хорошо программировать, но и обладать инженерным подходом к решению задач.

Мы собрали список материалов, которые могут помочь развить системный подход к разработке проектов.

Книги

Выделили необходимый минимум по программированию и архитектуре, добавили несколько книг о проектировании:

- Чистый код, Р. Мартин
- Чистая архитектура, Р. Мартин
- Читаемый код, Д. Бозвел, Т. Фаучер
- Рефакторинг, М. Фаулер
- Эффективная работа с легаси. М. Физерс
- В поисках требований, И. Александер
- Архитектура ПО на практике, Л. Басс, П. Клементс, Р. Кацман

Лекции

Собрали видеозаписи лекций об ООП, тестировании и гибком подходе к разработке:

- Principles of Object Oriented and Agile Design, Bob Martin
- The Principles of Clean Architecture, Bob Martin
- The Future of Progamming, Bob Martin
- All the Little Things, Sandi Metz
- Get a Whiff of This, Sandi Metz

Ресурсы и подходы

Отобрали самые жирные ресурсы с шаблонами проектирования, подходами к разработке и рефакторингу:

- Refactoring.guru
- Design Patterns for Humans
- Design Patterns Game
- Design Pattern Implementations in TypeScript
- Теория решения изобретательских задач
- Пример разработки «Крестиков-ноликов» по TDD

Концепции и инструменты проектирования

Нашли инструменты и методологии моделирования и проектирования систем:

- Унифицированный язык моделирования, UML
- Диаграммы потоков данных, DFD
- Entity-relationship модель
- Методология моделирования, IDEF
- Разработка через тестирование, TDD

Об авторах

Эту книжку написали:

- Саша Беспоясов — разработчик в 0+X, соавтор Тяжеловато, бывший преподаватель в Нетологии
- Артём Самофалов — ведущий фронтенд-разработчик в Social Discovery Ventures, бывший преподаватель в LoftSchool

...И контрибьюторы проекта ❤️

Помочь нам

Мы будем рады, если вы:

- расскажете о книге в соцсетях, блогах, телеграм-каналах;
- полайкаете её на Гитхабе.